Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado

Aceleración mediante GPU de un correlador DiFX para radioastronomía

Autor: Carlos Golvano Díaz

Tutor/es: Manuel Prieto Mateo, Javier González García

2021

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado
Aceleración mediante GPU de un correlador DiFX para radioastronomía

Autor: Carlos Golvano Diaz

Tutores: Manuel Prieto Mateo
Cotutor: Javier González García

TRIBUNAL:

Presidente: Agustín Martínez Hellín

Vocal 1º: Juan Ignacio Pérez Sanz

Vocal 2º: Eliseo García García

FECHA: 28 de septiembre de 2021

Dedicatoria

A mi padre y a mi madre.

Agradecimientos

A Javier González García por presentarme la idea de este Trabajo de Fin de Grado y por trabajar para poder llevarla a cabo, además de todo el conocimiento que me ha enseñado acerca del mundo de la radioastronomía y de VLBI. Sobre todo por la habilidad de explicar conceptos muy complejos de una forma tan sencilla.

A los compañeros del Observatorio de Yebes haciendo que mi estancia en el centro haya sido tan agradable.

A Pablo de Vicente Abad, director del Observatorio de Yebes, que ha permitido que pueda utilizar los recursos del observatorio para la realización de este proyecto.

A Manuel Prieto Mateo por aceptar este trabajo y posibilitar que se llevase a cabo.

A mis compañeros Carlos, Lucas y Miguel, que me han acompañado en estos cuatro años de carrera y que han sido un gran apoyo en los momentos más complicados.

Y por último, a mi madre y a mi padre por apoyarme y escuchar las charlas en las cenas sobre las cosas tan raras que aprendía en el observatorio. Y a mi tía Teresa por mostrarme el maravilloso mundo de la astronomía desde que era pequeño y que despertó gran interés en mi.

Índice general

Índice de figuras

Índice de tablas

Resumen

El campo de la radioastronomía, al igual que muchos otros ámbitos científicos está sufriendo una evolución acelerada, requiriendo cada vez mayores y mejores resultados en menor tiempo. En los últimos años, el creciente avance y la popularización de las tarjetas gráficas como elementos de computación para aplicaciones científicas está permitiendo el desarrollo de muchas áreas que estaban limitadas por la capacidad de cómputo, produciendo la aceleración antes mencionada. En este TFG se explora un avance hacia la utilización de tarjetas gráficas en las observaciones radio astronómicas de VLBI (Very Large Base Interferometry) con la intención de acortar el tiempo de correlación requerido para procesar los datos obtenidos. El Observatorio de Yebes, perteneciente al Instituto Geográfico Nacional, cuenta con el correlador software de código abierto DiFX, y es el sujeto de este estudio

Palabras clave: VLBI, correlador, GPU, Transformada Rápida de Fourier

Abstract

The field of radio astronomy, like many other scientific fields, is in constant growth, requiring more and better results in less time. In recent years, the advance of graphics cards as computing elements for scientific applications is allowing the development of many areas limited by computational capacity. This thesis explores an advance towards the use of graphics cards in Very Large Base Interferometry or VLBI radio astronmical observations with the purpose of improving the speed of correlations. The Yebes Observatory, which belongs to the National Geographic Institute, counts with DiFX, an open software correlator, which is the subject of this study.

Key words: VLBI, correlator, GPU, Fast Fourier transform

Acrónimos

- **EVN:** European VLBI Network
- **EU-VGOS:** European-VGOS
- **IVS:** International VLBI Service for Atrometry and Geodesy
- **VLBI:** Very Long Base Interferometry
- **VEX:** Archivo de definición de un experimento VLBI
- **VDIF:** Formato de los datos VLBI
- **FFT:** Fast Fourier Transform
- **CUDA:** Compute Unified Device Architecture

Definiciones

- **Conjuntos de interferómetros:** Conjunto de radiotelescopios individuales que trabajan de forma conjunta como un único telescopio.
- **Línea de base:** Vector formado entre dos radiotelescopios.
- **Host:** Al hablar de programación en tarjetas gráficas, referente a la CPU.
- **Device:** Al hablar de programación en tarjetas gráficas, referente a la GPU.

Capítulo 1

Introducción

1.1. Contexto

El Observatorio de Yebes es una Infraestructura Científica y Técnica Singular (ICTS) a 20 Km. al sur de Guadalajara. Pertenece al Instituto Geográfico Nacional (IGN) y está coordinado por la Subdirección General de Astronomía, Geofísica y Aplicaciones Espaciales. Es un centro de radioastronomía y geodesia en el que se llevan a cabo tareas de investigación y desarrollo tecnológico.

Cuenta con un pabellón de gravimetría, un radio telescopio con 13,2 metros de diámetro para aplicaciones geodésicas, otro radio telescopio de 40 metros para observaciones astronómicas y, en menor medida, aplicaciones geodésicas, y varios laboratorios dedicados al desarrollo de alta tecnología para la ciencia del espacio y la geodesia. Estas instalaciones incluyen un laboratorio de receptores criogénicos de radioastronomía, un laboratorio de electrónica y dispositivos de detección, un laboratorio de electroquímica, un laboratorio de medida (cámara anecóica), de alimentadores de microondas, y dos talleres con máquinas de precisión y mecanizado.

Entre las diversas tareas que se desarrollan en el centro, se encuentran las observaciones astronómicas y geodésicas utilizando la compleja técnica de Interferometría de Muy Larga Base (VLBI, *Very Long Baseline Interferometry*), que permiten obtener imágenes de radio con longitudes de onda del orden de centímetros y milímetros, con gran resolución (hasta decenas de microsegundos de arco). Con esta técnica también es posible estimar la distancia entre dos radio telescopios con una precisión por debajo de los centímetros. Esta capacidad tiene importantes aplicaciones en el campo de la geodesia espacial y la geofísica ya que permite, entre otras, controlar el movimiento de las placas continentales o materializar el Sistema de Referencia Celeste (ICRS, *Celestial, Reference System*) a través de fuentes extragalácticas.

1.2. Objetivo del proyecto

El objetivo principal del proyecto es modificar el actual correlador DiFX utilizado en el observatorio de Yebes para que la Transformada de Fourier se ejecute en una tarjeta gráfica. Con esto se pretende acelerar los tiempos de correlación, además de realizar un primer acercamiento al código y discutir los demás posibles cambios que se pueden realizar para que cada vez más partes sean posibles de ejecutar en una tarjeta gráfica.

En la actualidad aún no se han desarrollado apenas correladores para que utilicen tarjetas

gráficas, es por ello que este TFG pretende proponer ideas para que se vayan adaptando y desarrollando nuevos correladores y así mejorar el rendimiento de estos.

Capítulo 2

Correlador en VLBI

La técnica de Interferometría de Muy Larga Base o VLBI (Very Large Base Interferometry), desarrollada en la década de 1960 (Clark et al. 1967; Miran et al. 1967) consiste en obtener la densidad de potencia de la señal de radio captadas por dos o más radiotelescopios separados normalmente por cientos de kilómetros, consiguiendo una resolución mucho mayor respecto a la que consigue un radiotelescopio de forma independiente. El teorema de Weiner-Khinchin nos permite expresar dicha densidad de potencia en términos de una función de autocorrelación y que se puede aplicar a cada par de antenas. Para una explicación más detallada ver el apéndice A. Al resultado de este proceso se le denomina *Visibilidad*. Estas visibilidades tienen diferentes aplicaciones dependiendo de si se utilizan para astronomía o para geodesia.

En astronomía lo que se pretende es conseguir la mayor información posible sobre la fuente que se está observando. La Síntesis de Apertura, utilizada por primera vez por M. Riley y A. Hewish y que les llevó a ganar el premio Nobel de Física en 1974, es utilizada para la generación de los mapas de estas fuentes de radio y que además permite, haciendo uso de una red de radiotelescopios, obtener una resolución angular igual a la que tendría una sola antena del tamaño de la línea de base del par de antenas más distantes dentro de la red. Sin embargo, como veremos más adelante, para conseguir una correlación correcta de las señales de las diferentes antenas y las consiguientes visibilidades, es necesario corregir el retardo con el que los radiotelescopios captan la señal del cielo.

La geodesia, a diferencia de la astronomía, no se interesa por la información del cielo. Sino que el objetivo de estas observaciones es calcular el retardo con el que llega la señal a las diferentes antenas, con lo que se puede calcular la distancia entre estas.

En la realización de un experimento varias antenas apuntan al mismo punto en el cielo recibiendo la misma señal de radio. En la figura 2.1 se puede ver un sencillo esquema de un interferómetro VLBI compuesto por dos antenas. Las antenas están separadas por cientos o miles de kilómetros, por lo que cada una estará a una distancia diferente de la fuente observada. Esta diferencia representa un retardo respecto al momento en el que la señal llega a cada una de las antenas. La primera impresión suele ser que esta diferencia entre las antenas podría ser despreciable, teniendo en cuenta las magnitudes astronómicas (por ejemplo, años luz). Sin embargo, cuando se trabaja con señales electromagnéticas y con la precisión que requiere la radioastronomía, unos cuantos kilómetros puede suponer una gran diferencia en los resultados del experimento.

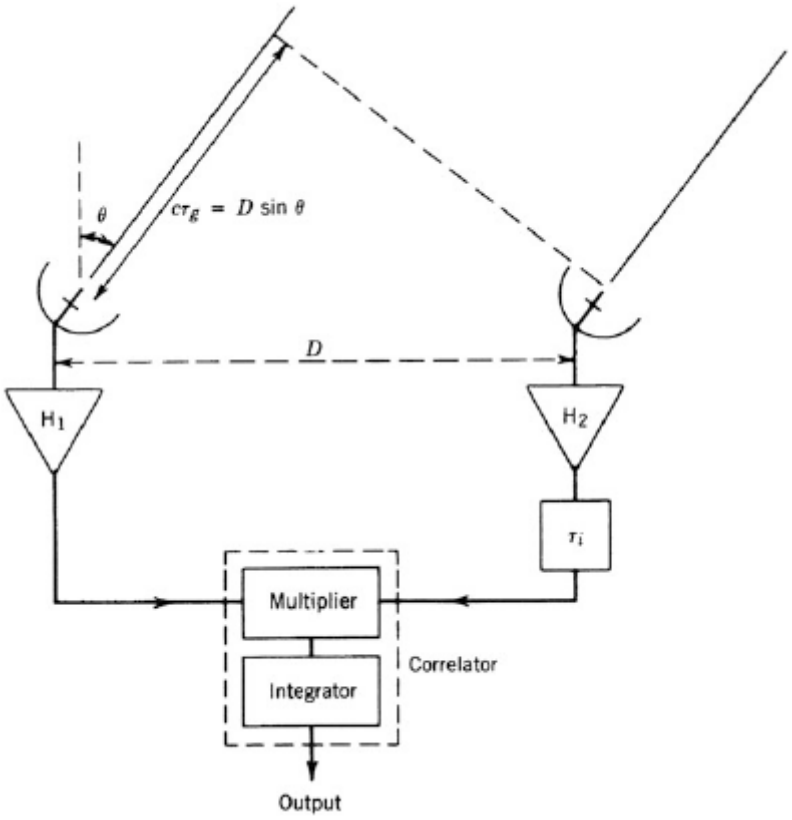


Figura 2.1: Esquema sencillo de un interferómetro VLBI

Este retardo se define como la derivada de la fase respecto a la frecuencia (ecuación 2.1), y que corresponde a un máximo en la función de correlación, denominado franja. Este retardo se denomina como *retardo de grupo* (*group delay*).

$$\tau_{gd} = \frac{\partial \phi}{\partial \omega} \tag{2.1}$$

Debido a la rotación de la tierra, la distancia entre las antenas y la fuente va variando y por consiguiente el retardo calculado. Esto hace que no podamos utilizar el mismo retardo durante toda la correlación, ya que el máximo de correlación se va desplazando del punto inicial. Este efecto se denomina *tasa de retardo* y se define como la derivada del retardo de grupo respecto al tiempo (ecuación 2.2).

$$\tau = \frac{\partial \tau_{gd}}{\partial t} \tag{2.2}$$

El correlador es capaz de conocer la tasa de retardo y compensarlo, introduciendo un retardo τ_i, denominado *retardo instrumental*.

En geodesia el objetivo es calcular la tasa de retardo. Una vez se ha determinado, mediante trigonometría se puede calcular la distancia que separa a las dos antenas que han recibido la señal, formando una línea de base, como se puede ver en la figura 2.2.

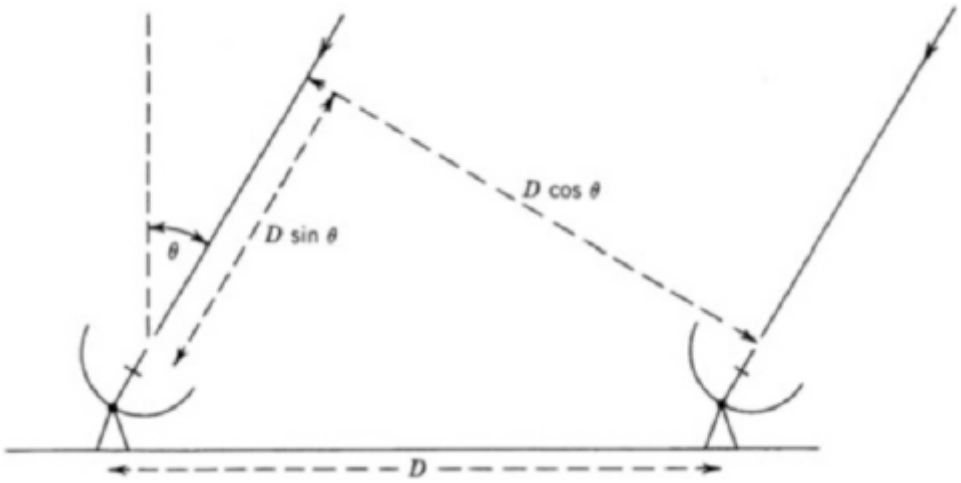


Figura 2.2: Trigonometría de un modelo clásico de VLBI

En las observaciones astronómicas es importante conocer la línea de base entre dos antenas con precisión para poder compensar la tasa de retraso con el retraso instrumental y obtener buenos resultados en la correlación. Además, conocer la línea de base tiene otras aplicaciones como la monitorización del movimiento de placas tectónicas.

Por último, una parte importante es la sincronización de los relojes de cada antena, debido a que las muestras llevan marcas temporales que permitirán alinearlas más tarde. Estas marcas deben ser exactamente las mismas para todas las antenas, lo que supone una precisión de los relojes bastante alta. Es por ello que se utiliza la hora que emite la constelación GPS como referencia. Los mensajes que envían los satélites llegan al mismo tiempo a todas las antenas y va incluida, entre otras cosas, una hora igual para cualquier mensaje de cualquier satélite.

Capítulo 3

Necesidad de correladores más rápidos

La demanda computacional en la investigación científica está en constante aumento, requiriendo sistemas y máquinas más rápidas y mejor optimizadas. En el mundo de la radioastronomía también existe esta evolución hacia sistemas computacionales más complejos, pasando de observaciones con una sola antena a conjuntos de interferómetros. A su vez, la tecnología de VLBI ha aumentado la precisión de las observaciones, adquiriendo un rápido desarrollo. Este aumento de precisión normalmente implica un mayor número de antenas por observación y una mayor tasa de grabado, generando más datos por observación y mayor requerimiento computacional a la hora de realizar la correlación.

Desde el inicio, se ha utilizado hardware dedicado para realizar las correlaciones. Entre otros, los más empleado han sido los circuitos integrados de aplicación específica (ASIC, *Application Specific Integrated Circuit*), procesadores de señales digitales (DSP, *Digital Signal Processing*) y matrices de puertas lógidas programables en campo (FPGA, *Field-programmable Gate Array*). La elección de qué tecnología utilizar depende de las necesidades del proyecto. Mientras que DSP es más adecuado para tareas de procesamiento computacional, FPGA logra mejores resultados con operaciones lógicas. Un ejemplo de correlador basado en DSP es el implementado por A. R. Whitney et al. [1]. Otro ejemplo de correlador basado en FPGA es el diseñado por Rurik A. Primiani et al. [2], que reemplazaría a los correladores ASIC. Por último, tenemos el trabajo realizado en el programa RadioNet FP7, donde se diseñó un correlador de alto rendimiento basado en DSP y FPGA, llamado UniBoard [3]. Aunque estos métodos de hardware dedicado son bastante eficientes, tienen grandes inconvenientes como el costo económico o su escasa escalabilidad.

Recientemente ha surgido una nueva tendencia que consiste en utilizar software que realice las correlaciones, en vez de hardware. El principal objetivo es solucionar los problemas de coste y escalabilidad de los correladores basados en chips dedicados. El desarrollo de estos nuevos correladores se apoyan en el rápido avance de la tecnología computacional que estamos viviendo en las ultimas décadas.

Los primeros que se desarrollaron utilizaban únicamente una CPU como unidad de procesamiento, por ejemplo, el correlador del Observatorio Nacional de Radio Astronomía (NRAO, *National Radio Astronomy Observatory*) fue desarrollado basándose en un IBM 360/50 [4]. Sin embargo, rápidamente se pasó a utilizar clústers de CPUs para conseguir mayor eficiencia, co-

mo la Red China de VLBI (CVN, *Chinese VLBI Network*) desarrollado por el Observatorio Astronómico de Shanghai de la Academia de Ciencias China [5]. Otro ejemplo de correlador basado en un clúster es el correlador DiFX para VLBI propuesto por la Universidad de Tecnología de Swinburne [6]. Este último es importante ya que será el que se utilice en este TFG. Desde el inicio de esta tecnología hasta los último años, todos los correladores se han diseñado para tener que utilizar únicamente la CPU.

En los últimos años las tarjetas gráficas (GPU, *Graphics Processing Unit*) han evolucionado a gran velocidad, mejorando sus especificaciones de forma considerable cada año. Debido a esto, actualmente se están empezando a desarrollar los primeros correladores basados en GPU. Las tarjetas gráficas se componen de un gran número de núcleos que son capaces de ejecutar procesos sencillos. Estos núcleos pueden ser ejecutados de forma paralela, componiendo un sistema multihilo y una gran capacidad de paralelización. Hasta hace unos años únicamente se estudiaba la viabilidad de crear un correlador basado en GPU. Entre estos estudios podemos encontrar el de Thomas Hobiger et al. [7]. Sin embargo, hemos tenido que esperar hasta este último año 2021 para poder ver un correlador para VLBI basado en GPU totalmente funcional, desarrollado por F. Zhang et al. [8].

El escenario que se busca en el ámbito de los correladores es que sean capaces de procesar los datos en tiempo real y proporcionar los resultados y las visibilidades a la vez que termina de realizarse la observación. Actualmente esto no es posible hacerlo en la mayoría de los casos debido a las limitaciones computacionales de los correladores, no son suficientemente rápidos, y las impuestas por la infraestructura de las redes de comunicación, que no poseen una velocidad de transmisión de datos lo suficientemente rápida. Lo que se pretende al desarrollar correladores para que funcionen sobre GPUs en vez de CPUs es solucionar las limitaciones computacionales, logrando tiempos más pequeños a la hora de hacer las correlaciones.

Capítulo 4

Funcionamiento de una GPU

Para poder comprender la razón por la que actualmente se están empezando a diseñar correladores para tarjetas gráficas en vez para procesadores es necesario entender la diferencia entre ambos componentes y en qué casos es preferible utilizar uno frente al otro.

Por un lado, la CPU es el elemento principal de un computador, que se encarga de ejecutar los procesos y de dar instrucciones al resto de componentes. Es un chip, también denominado procesador, que se compone de uno o varios núcleos, que normalmente no superan los 8. Cada núcleo es capaz de ejecutar uno o dos hilos, que se encargan de ejecutar los diferentes procesos. El número de hilos que ejecuta cada núcleo dependerá de la arquitectura del procesador. Debido a que la CPU es el elemento que controla el computador, su arquitectura es bastante compleja, permitiendo que pueda ejecutar tareas complejas a nivel computacional como la recursividad.

Por el otro lado, la GPU se compone de una gran cantidad de núcleos capaces de ejecutar procesos sencillos. El número de núcleos depende de la tarjeta gráfica, se puede ver una lista de diferentes tarjetas gráficas en la tabla 4.1. La diferencia entre los núcleos de la GPU y una CPU es que estos tienen una arquitectura más sencilla y no son capaces de ejecutar procesos complejos. La ventaja la encontramos en que pueden funcionar de forma simultanea, consiguiendo un nivel de paralelización muy alto en comparación con la CPU. Uno de los casos en los que la GPU tiene bastante utilidad es en las operaciones con matrices, donde hay que realizar muchas operaciones matemáticas sencillas. Por ejemplo, a la hora de sumar dos matrices la CPU lo hará de forma secuencial, hasta que no acabe una suma no empieza otra, mientras que la GPU lo hará de forma paralela, cada núcleo se encarga de una suma y se hacen todas al mismo tiempo.

GPU	Núcleos (CUDA cores)	Memoria	Frecuencia del procesador
GeForce RTX 3090	10496	24 GB	1395 / 1695 MHz
GeForce GTX 1080	2560	8 GB	1607 / 1733 MHz
GeForce GTX 780	2304	3 GB	863 / 900 MHz
Quadro RTX 8000	4608	48 GB	1395 / 1770 MHz
Quadro M2000	768	4 GB	796 / 1163 MHz

Tabla 4.1: Especificaciones de diferentes tarjetas gráficas

Aparte de las ventajas que ofrece la GPU existe un gran inconveniente que suele ser el limitante a la hora de mejorar los tiempos de computación. En un computador, inicialmente los datos se encuentran alojados en la memoria, sin embargo, la GPU necesita tenerlos en su

propia memoria. Es por ello que hay que moverlos de la memoria principal a la memoria de la GPU y esta suele ser la operación que consume el mayor tiempo. A la hora de diseñar sistemas que utilicen tarjetas gráficas, lo que se pretende es realizar esta transmisión de datos la menor cantidad de veces posible, que se consigue manteniendo la información en la memoria de la tarjeta gráfica y únicamente pasarla a la memoria principal cuando sea estrictamente necesario.

Para poder manejar una tarjeta gráfica necesitamos unas herramientas que nos permitan comunicarnos con ella y darle instrucciones. Actualmente tenemos CUDA (*Compute Unified Device Architecture*), plataforma de computación en paralelo que incluye un compilador y un conjunto de herramientas de desarrollo. Esta plataforma pertenece a Nvidia, empresa dedicada a la creación de tarjetas gráficas. A partir de este punto, tomaremos las tarjetas gráficas de Nvidia como referencia, debido a que CUDA únicamente se puede utilizar en éstas.

Capítulo 5

Planteamiento del proyecto

5.1. Origen

La astronomía siempre ha sido un campo de gran interés para mi, es por ello que decidí realizar las prácticas curriculares en el observatorio de Yebes. Durante este periodo, vi que uno de los proyectos que se querían llevar a cabo era el de instalar un correlador en el propio observatorio para una futura red de radio telescopios. Además, también se quería ver la viabilidad de adaptar dicho correlador para que fuese capaz de hacer uso de GPU.

A lo largo de la carrera he adquirido conocimientos de programación en tarjetas gráficas que junto a mi creciente interés en el mundo de la radio astronomía hicieron que decidiese proponer realizar la parte de analizar la viabilidad de un correlador en GPU como mi TFG.

5.2. Herramientas utilizadas

En este apartado se describe todo lo que ha sido necesario para la realización del proyecto. Encontramos cuatro grupos: control de versiones, hardware, correlador y lenguajes de desarrollo y librerías.

5.2.1. Control de versiones

Un controlador de versiones permite llevar un registro del trabajo realizado, permitiendo ver fácilmente los cambios entre las diferentes versiones y proporcionando una fuerte capacidad de reponer archivos borrados por accidente. La característica que más interesa para este proyecto es la seguridad que da el poder recuperar cualquier archivo registrado, evitando perder el trabajo realizado por accidente.

Posiblemente, la herramientas más utilizada para esta tarea es Git y es el que se ha decidido utilizar, en conjunto con GitHub.

5.2.2. Hardware

El hardware utilizado se divide en dos partes: el clúster construido en el observatorio de Yebes donde está montado el correlador y la GPU añadida a este clúster.

- Clúster

El actual correlador de Yebes está montado en un sistema distribuido de tres servidores comunicados por una red privada de 1 Gbps de velocidad (Figura 5.1).

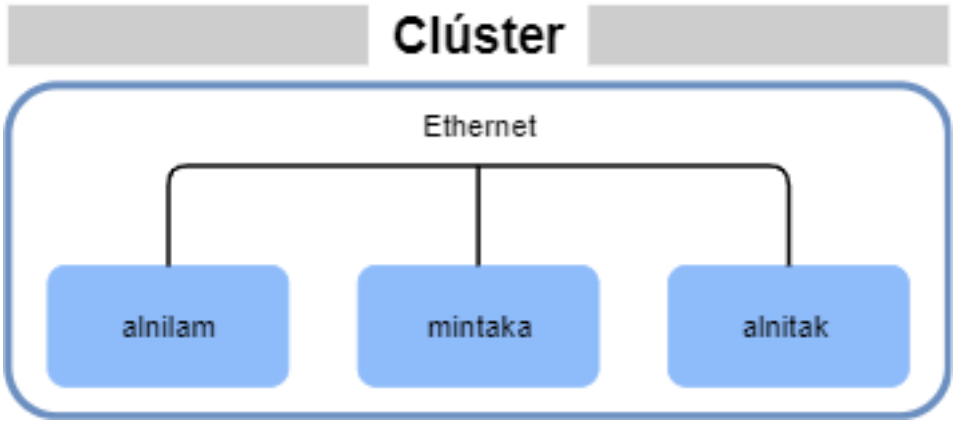


Figura 5.1: Diagrama del clúster formado por los tres servidores alnilam, mintaka y alnitak

Las especificaciones de estos equipos se pueden ver en la tabla 5.1.

Hostname	Nombre	CPU	No. Cores	No. Sockets	Hyper threading	RAM
alnilam	HPE ProLiant Gen 10	Intel Xeon Silver 4110 @ 2.10GHz	8	1	Yes	128 GB
alnitak	HPE ProLiant Gen 9	Intel Xeon(R) CPU E5-2620 v4 @ 2.10GHz	8	1	Yes	16 GB
mintaka	Supermicro XL	Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz	8	2	Yes	32 GB

Tabla 5.1: Especificaciones de los nodos para el clúster del Observatorio de Yebes

- GPU

El componente que se ha añadido a este sistema es la GPU. En este caso tenemos una Nvidia Quadro M2000 cuyas principales características se pueden ver en la tabla 5.2.

Nvidia Quadro M2000	
CUDA cores	768
Ancho de banda de la memoria	106 GB/s
Tipo de memoria	GDDR5
Memoria	4096 MB
Reloj base	796 MHz
Reloj acelerado	1163 MHz
Reloj de la memoria	1653 MHz 6.6 Gbps efectivos
Precio	460 €

Tabla 5.2: Especificaciones de Nvidia Quadro M2000

5.2.3. Correlador

El correlador que se ha empleado para este proyecto es DiFX, desarrollado por Deller, Tingay, Bailes & West en 2007 [6]. En el capítulo 6 se hace una explicación detallada de este.

5.2.4. Lenguajes de desarrollos y librerías

Los lenguajes utilizados en el proyecto han sido C++ y Python. También se ha utilizado CUDA que tiene como lenguaje de programación una variación de C++.

El correlador está escrito en su mayor parte en C++, por lo que este ha sido el más utilizado. Este lenguaje es una extensión de C, lenguaje de bajo nivel muy popular, que añade diferentes paradigmas como programación orientada a objetos y programación estructurada. Sin embargo, sigue estando presente el cercano manejo de la memoria característico de C. Una de las principales ventajas de este lenguaje es, aparte de la posibilidad de controlar el estado de la memoria, la gran velocidad con la que se ejecutan los procesos. La cantidad de librerías que ofrece C++ es bastante amplia, siendo muchas de estas para funciones básicas, por ejemplo, cstdlib. Aquí se van a exponer únicamente las librerías más relevantes.

- Intel Integrated Performance Primitives (*Intel ipp*)

  Intel Integrated Performance Primitives (Intel IPP) es una librería que ofrece una amplia gama de funcionalidades, como el procesamiento general de señales e imágenes, la visión informática, la compresión de datos y la manipulación de cadenas.

  Está optimizada por Intel para que se consiga el mayor rendimiento posible del procesador, utiliza variables propias y funciones de manejo de memoria propias. En DiFX se usa para hacer todas las operaciones matemáticas como suma, multiplicación, copia o borrado de vectores entre otras, incluyendo a la FFT. Las funciones encargadas de la Transformada de Fourier han sido sustituidas por las funciones de cuFFT.

- CUDA Fast Fourier Transform (*cuFFT*)

  Librería desarrollada por Nvidia que proporciona una sencilla interfaz para llevar a cabo la FFT en una tarjeta gráfica de Nvidia y está altamente optimizada y probada. Es la que sustituye a Intel ipp en la tarea de la Transformada de Fourier.

- Cuda_runtime y cuda

  Incluyen todas las funciones de CUDA necesarias para utilizar la GPU, como el manejo de la memoria de esta y el paso de datos entre host y device entre otros.

Python se ha utilizado para el análisis de los datos de las diferentes pruebas realizadas y la creación de gráficas para representarlos. Es un lenguaje sencillo, cómodo, bastante limpio e incluye librerías muy buenas para el manejo y análisis de datos. Es por ello por lo que se eligió para esta tarea. Entre todas las librerías utilizadas, las más relevantes son las siguientes:

- Numpy

  Proporciona soporte para el manejo de vectores y matrices multidimensionales de gran tamaño, además de gran cantidad de funciones matemáticas que facilitan ciertas tareas.

- Matplotlib

  Permite la visualización de datos y trazado de gráficos para Python. Como tal, ofrece una alternativa viable de código abierto a MATLAB.

Cabe destacar que también se ha utilizado BASH para la realización de las pruebas.

5.3. Planificación

1. Aprendizaje

   El objetivo fue obtener toda la información necesaria para el correcto entendimiento de como funciona un correlador FX para VLBI. Este apartado de divide en dos partes.

   a) Aprendizaje sobre los correladores para observaciones VLBI.

   b) Últimos avances en la implementación de la GPU en los correladores para VLBI.

2. Análisis del correlador DiFX

   Consistió en entender la arquitectura y el funcionamiento del correlador DiFX. También se identificaron todos los archivos y todo el código que sería necesario modificar.

3. Desarrollo del software

   Una vez se tuvo claro las partes a cambiar y como serían esos cambios se pasó a buscar las librerías necesarias y al diseño de dichos cambios.

   a) Identificación de las librerías a utilizar

   Se buscaron las librerías de CUDA que satisficieran los requisitos para las modificaciones, además de leer y entender la documentación tanto de la librería de Intel (Intel ipp) como la de CUDA (cuFFT). También se realizaron diferentes pruebas para comparar el rendimiento y el funcionamiento de ambas.

   b) Diseño de las modificaciones del código

   Una vez analizado el código, se diseñaron los cambios que se iban a realizar en este.

   c) Desarrollo e implementación de las modificaciones

   d) Pruebas del nuevo código

   Se realizaron pruebas para comprobar el correcto funcionamiento del nuevo correlador.

4. Obtención de las conclusiones

   Con todas las pruebas hechas y los datos recolectados se analizaron para la obtención de las conclusiones.

Capítulo 6

Correlador DiFX

Desde principios de los años 2000 y hasta la actualidad se han ido desarrollando gran cantidad de correladores software, sustituyendo a los anteriores que utilizaban hardware dedicado. Esta tendencia se ha ido incrementando con el rápido avance de la tecnología computacional. La principal razón de este cambio es lo poco manejables que eran los antiguos correladores. Eran construidos para tipos de observaciones específicas y si había algún cambio en la manera de observar era muy difícil adaptarlos a esos cambios. Además de que el coste de estos computadores era muy alto.

Sin embargo, los correladores que se hacen utilizando software en vez de hardware no tienen estos problemas. Pueden funcionar en prácticamente cualquier máquina, inclusive en un ordenador personal, son sencillos de modificar y adaptar a los cambios en la manera de realizar los experimentos, el coste de crearlos es más bajo que el de crear un correlador utilizando hardware y, si es código abierto, la posibilidad de que otros desarrolladores puedan aportar ideas o incluso código desde otras partes del mundo.

DiFX fue desarrollado en la Universidad de Tecnología de Swinburne, por A.T. Deller, S.J. Tingay, M. Bailes & C.West [6] en 2007. Como su nombre indica, es un correlador FX, que indica el método utilizado para realizar la correlación.

Proporciona un motor que realiza la correlación, así como diversas herramientas para manejarlo y monitorizarlo. También se incluye software adicional que ayuda con la depuración de errores, organización de los archivos y una interfaz para los diferentes datos que generan las observaciones de VLBI. Está escrito en C/C++ en su mayoría, haciendo un gran uso de la librería Intel IPP y, actualmente, enfocado para trabajar en sistemas con sistemas operativos Linux. Existe una wiki con la documentación detallada de la arquitectura y el funcionamiento del correlador:

https://www.atnf.csiro.au/vlbi/dokuwiki/doku.php/difx/start

En los siguientes apartados se describen los fundamentos de cualquier correlador FX, así como el de DiFX.

Varias de las operaciones iniciales son basadas en los telescopios mientras que algunas de las operaciones siguientes son basadas en la línea de base. Estos dos conjuntos de operaciones se describen brevemente por separado y en secuencia.

6.1. Operaciones basadas en las antenas

6.1.1. Alineamiento de los datos

Este es el primer paso que hace un correlador y consiste en aplicar una primera corrección a las señales para alinearlas. La señal recibida por la antena se graba como un flujo de datos, de forma que esta señal está ordenada en función del tiempo en el que alcanzó la antena. Este flujo es el que toma el correlador como entrada.

Con el software Cacl9, se introduce el retardo instrumental τ_i para compensar el retardo geométrico y acercar la franja de correlación a 0.

Calc9 genera un modelo de retardo geométrico $\tau(t)$ para cada antena en intervalos ajustables (1 segundo por defecto). También tiene en cuenta otros retardos no geométricos como la precesión, la nutación o la carga oceánica y atmosférica.

Los datos de cada telescopio son almacenados en buffers en memoria y se irán alineando. Este alineamiento se realiza teniendo en cuenta que la antena se encuentra en el centro geométrico de la Tierra, teniendo así una referencia común para todas las antenas participantes en el experimento. En este punto, se lleva a cabo la primera alineación, llamada *integer-sample delay* o *retardo de muestra entera*. Esta corrección se realiza a nivel de muestra por lo que no es del todo preciso. La diferencia entre el retardo teórico y el retardo de muestra entera se almacena como *fractional sample delay* o *retardo fraccionario de la muestra*. La ecuación 6.1 muestra cómo se corrige el retardo de muestra entera y como se calcula el retardo fraccionario.

$$\tau_t - N\Delta t = \epsilon \tag{6.1}$$

Donde τ_t es el retardo teórico, Δt es la frecuencia de muestreo en segundos y N la cantidad de tiempos de muestreo que se a va desplazar, por ello la precisión de este alineamiento es de ± 1 muestra. Por último, ϵ sería el retardo que falta por corregir, es decir el retardo fraccionario.

La corrección se realiza desplazando el puntero de memoria la cantidad de muestras correspondientes. Una vez los retardos han sido ajustados el formato de los datos habrá cambiado, pasando de una representación de, normalmente, 2 bits a una de coma flotante. Esta transformación es posible realizarla sin ninguna pérdida de información gracias al teorema de Van-Vleck.

6.1.2. Rotación de franja

Con las señales alineadas se podría pensar que la franja de correlación ya es cero o muy cercana a cero. Sin embargo, si hiciésemos la correlación únicamente con dicho alineamiento no encontraríamos la franja, quedando una correlación errónea. El problema viene del momento temporal en el que se están aplicando los retardos a las diferentes señales.

Para explicarlo vamos a tomar dos antenas, A_1 y A_2 que forman una línea de base. Estas dos antenas están apuntando a la misma fuente por lo que están recibiendo la misma señal (figura 6.1). La señal del cielo o señal de radiofrecuencia ν_{rf} llega a la antena A_2 con un retardo τ_g respecto a la antena A_1. Este retardo ocurre de forma natural debido a que las antenas no están a la misma distancia de la fuente.

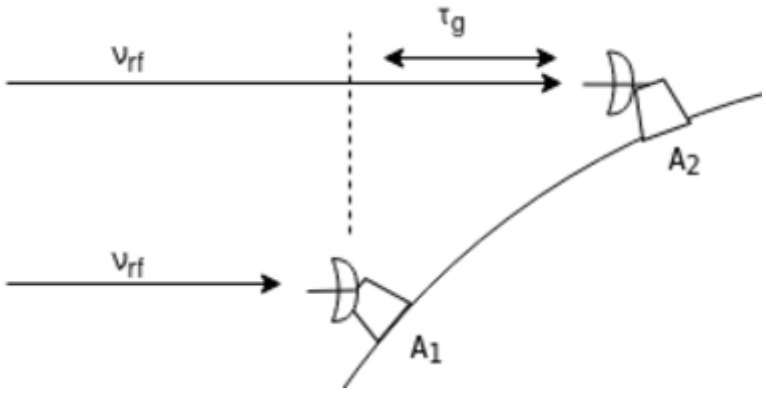


Figura 6.1: Esquema de antenas recibiendo una señal del cielo

El retardo τ_g se compensa más tarde añadiendo un retardo instrumental τ_i a la antena A_1. Aunque a primera instancia parece que no hay ningún problema y que las dos señales deberían estar alineadas, hay un problema que viene de aplicar el retardo instrumental a una frecuencia distinta a la que se aplica el retardo geométrico τ_g.

En las antenas, la señal recibida es procesada para desplazarla a frecuencias menores, siguiendo el esquema de la figura 6.2. El desplazamiento se consigue mezclando la señal del cielo con una señal ν_{LO} producida por el oscilador local (*local oscilator*). El resultado es una señal ν_{IF} disminuida en frecuencia, denominada señal de frecuencia intermedia (*intermediate frequency*), cuya frecuencia es diferente a la de la señal del cielo ν_{rf}. Las ecuaciones 6.2 y 6.3 corresponden a las señales de los telescopios A_1 y A_2 respectivamente.

$$F_1 = A_1 cos(2\pi \cdot \nu_{IF} \cdot \tau_i) \tag{6.2}$$

$$F_2 = A_2 cos(2\pi \cdot (\nu_{LO} + \nu_{IF}) \cdot \tau_g) \tag{6.3}$$

En A_1 el retardo τ_i se está aplicando sobre ν_{IF}, mientras que en A_2 el retardo τ_g se aplica sobre la señal del cielo ν_{rf}, que corresponde a $\nu_{LO} + \nu_{IF}$. Este problema no existiría si el retardo instrumental se aplicase a la señal del cielo, es decir, nada mas captarla. Sin embargo, esto es impracticable.

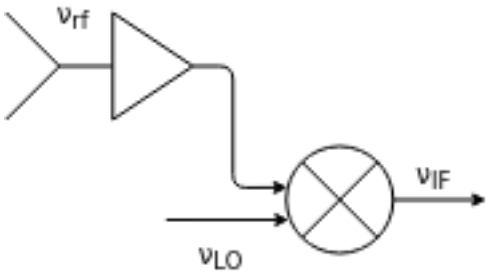


Figura 6.2: Esquema simplificado de la disminución de frecuencia de la señal del cielo

La corrección de franja se puede realizar antes de la FFT (pre-F) o después (post-F). En el caso de pre-F la corrección se hace en el dominio del tiempo con una translación en el tiempo, haciendo uso de la propiedad de la transformada de Fourier descrita en la ecuación 6.4.

$$\mathbb{F}[t - t_0] = F(\omega)e^{-j\omega t_0} \tag{6.4}$$

Para el caso de post-F se realiza una translación en la frecuencia, aplicando otra de las propiedades de la transformada de Fourier (ecuación 6.5).

$$\mathbb{F}[f(t)e^{j\omega_0 t}] = F(\omega - \omega_0) \tag{6.5}$$

6.1.3. Canalización y corrección del retardo fraccionario de la muestra

La canalización no es otra cosa que realizar la FFT por tramos. Ya se ha explicado que no es viable hacer la FFT de toda la señal original debido a que los retardos aplicados para alinear las señales no es fijo, sino que varía, además de la complejidad de hacer una FFT tan grande. Es por ello que se divide la señal original para integrarla en intervalos definidos por el periodo de subintegración (suele ser de 1 segundo). La integral que abarca el periodo de subintegración también se divide en integrales más pequeñas que abarquen el número de puntos configurado. Son estas FFTs las que se denominan canalización.

La última operación que se realiza sobre los datos de cada antena es la corrección del retardo fraccionario de la muestra, que habíamos almacenado en el apartado de alineamiento de los datos. Se puede aplicar haciendo una multiplicación compleja con las muestras en el dominio de la frecuencia o, si se ha escogido la opción de post-F para la corrección de la franja, es tan simple como añadir el retardo a la exponencial compleja.

6.2. Operaciones de la línea de base

6.2.1. Multiplicación cruzada

Una vez se ha hecho la FFT y las correcciones de los retardos correspondientes solo queda multiplicar los datos de cada línea de base muestra por muestra, habiendo antes conjugado una de ellas. En los casos que los datos estén polarizados, se hacen cuatro multiplicaciones, de cada polarización con el resto.

6.2.2. Integración

En la práctica no se suele realizar una sola integral que abarque el periodo de integración. Normalmente se divide en varias partes en función de la resolución que se haya configurado y se hace la FFT de estas. Finalmente se acumulan todas obteniendo el resultado para ese tiempo de integración.

6.3. Software del correlador DiFX

Una vez entendidos los principales pasos que lleva a cabo DiFX para correlar los datos de dos o más antenas participantes en un mismo experimento, se puede pasar a explicar el software que los lleva a cabo. Se va a explicar la estructura, como se opera y los archivos más relevantes, haciendo especial énfasis en los que han sufrido las modificaciones.

6.3.1. Estructura

El correlador se compone de tres partes (figura 6.3) que realizan diferentes funciones y que, normalmente, se ejecutan en diferentes computadores.

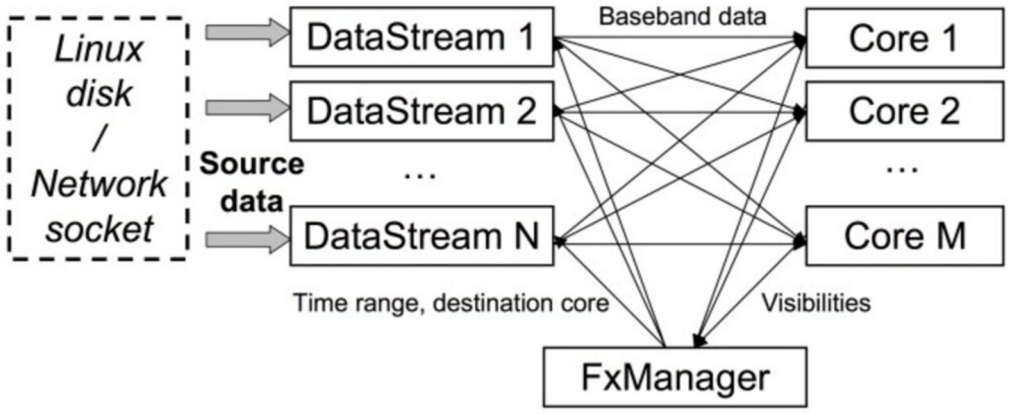


Figura 6.3: Arquitectura de un correlador DiFX

- **FxManager:** Ejecutado en un sólo computador, es el encargado de sincronizar las tareas de los DataStreams y de los Cores, asignando trozos, o chunks, de datos, configurados por el periodo de subintegración, que deben leer los DataStreams y enviar a los Cores. Una vez hecha la correlación las visibilidades se envían a este nodo que se encarga de guardarlas en el disco.

- **DataStream:** La función de esta parte es leer los datos del disco y guardarlos en un buffer circular del tamaño del número de subintegraciones que se hayan configurado. Suele haber más de un hilo encargado de esta tarea por nodo, dependiendo de la capacidad de paralelización del procesador y de la cantidad de servidores disponibles hay para esta función. Esto se consigue utilizando el paradigma de envío de mensajes acorde a la Interfaz de Envío de Mensajes (MPI, *Message Passing Interface*).

- **Core:** Todas las operaciones de la correlación, como la FFT y la corrección de los retardos, se llevan a cabo en esta parte. Únicamente podemos tener un nodo por cada computadora, por lo que la cantidad de nodos de este tipo que se utilicen dependerá de la cantidad de servidores disponibles.

  Cada nodo puede llevar a cabo cuatro subintegraciones. Las subintegraciones son procesadas por N hilos que se encargan de procesar 1/N de los datos. Es por esto que no se realiza una sola integral de todo el periodo de subintegración, sino que se divide en función de la cantidad de hilos que se generen. Se puede ver la estructura del Core de forma detallada en la figura 6.4

  Una vez lanzado el Core se queda en un bucle a la espera de instrucciones por parte del FxManager del intervalo de tiempo que debe correlar, después recibe los datos necesarios por parte del DataStream y, finalmente, lanza los hilos para procesar la subintegración correspondiente.

En el observatorio de Yebes hay tres servidores que forman el clúster y que se dividen cada una de estas tres partes. La función de FxManager la realiza alnilam, además de encargarse de los DataStreams, del Core se encargan alnitak y mintaka.

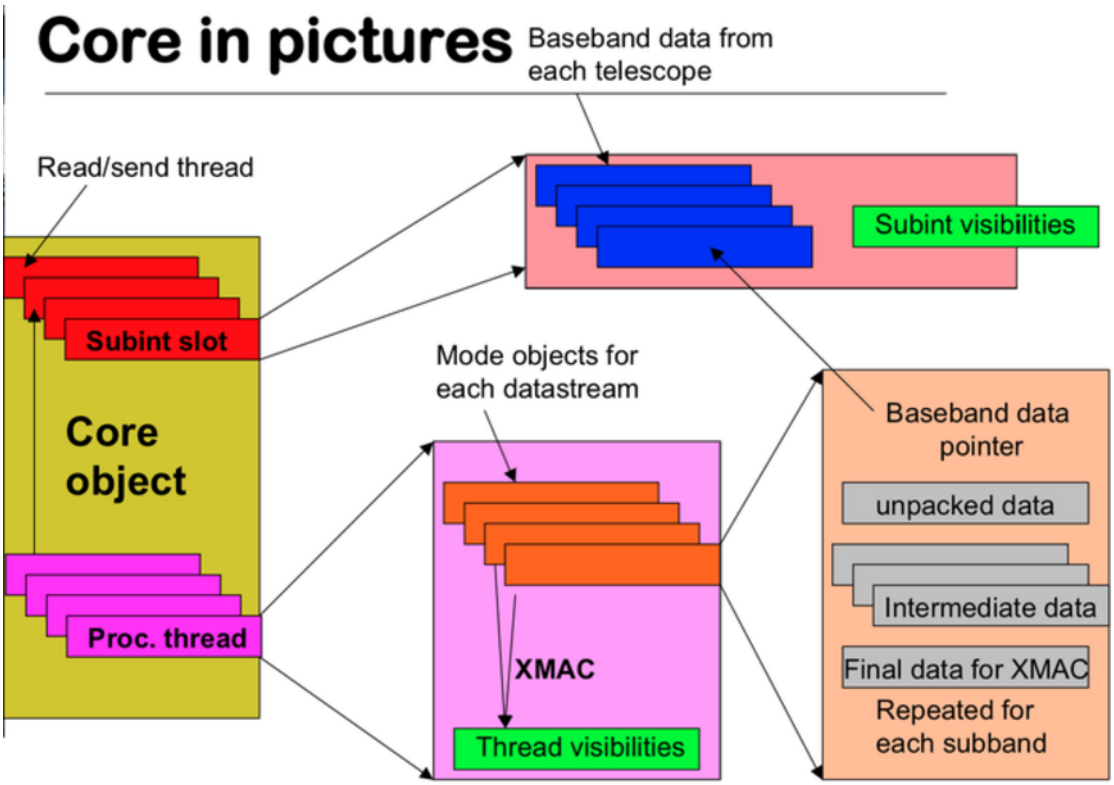


Figura 6.4: Estructura del Core de DiFX

Después de adaptar el correlador para el propósito de este TFG la configuración cambió. Debido a que sólo se disponía de una GPU y que estaba montada en alnitak, este fue el que llevó a cabo la función de Core. Por otro lado, mintaka dejó de usarse y alnilam siguió con las mismas funciones.

6.4. Operación de DiFX

El control de DiFX se realiza mediante archivos de configuración, que el operador configura previamente con las opciones que mejor se adapten a la correlación. Una vez que se ha lanzado no hay ninguna manera de modificar dicha configuración, es decir que el operador ya no tiene ninguna interacción con DiFX. Normalmente el operador solo necesita editar dos archivos, a partir de los cuales se generarán el resto. Estos son el archivo VEX que contiene toda la información para realizar el experimento y el archivo V2D desde el que se pueden especificar gran cantidad de parámetros. La figura 6.5 es un diagrama de flujo que representa los pasos a seguir hasta tener todos los archivos necesarios. Las cajas naranjas son los pasos que el operador tiene que hacer manualmente mientras que las amarillas son los pasos que se hacen de forma automática. La parte de la izquierda, con las letras en rojo, son los archivos de entrada que requiere ese proceso, mientras que la parte de la derecha, con letras en azul, son los ficheros de salida del proceso.

6.4.1. Archivo de configuración (V2D) y archivo de descripción de un experimento VLBI (VEX)

El archivos de descripción de un experimento VLBI (.vex) contiene toda la información necesaria para llevar a cabo una observación. Contiene, entre muchas cosas, las antenas que participan y su información, la lista de todas las fuentes que se van a observar y el momento

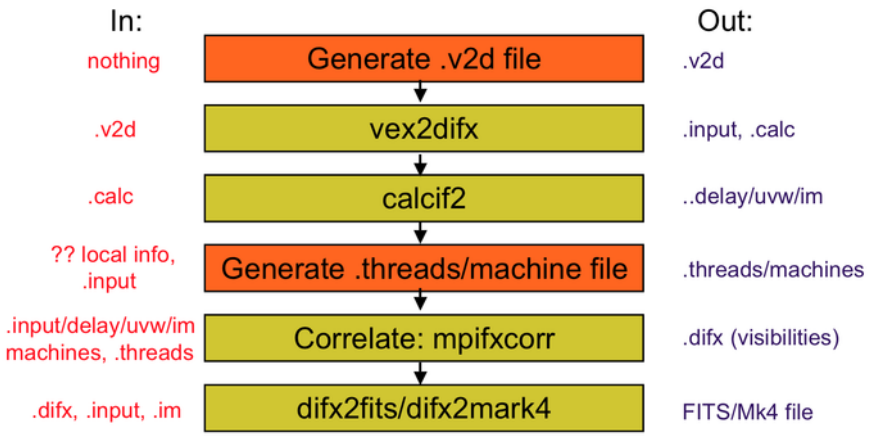


Figura 6.5: Diagrama de flujo de la generación de archivos para el funcionamiento de DiFX

exacto en el que tienen que empezar a grabar los datos, el rango de frecuencias en el que se va a observar y la cantidad de canales. Debido a que este archivo es el que se utiliza para la observación, a la hora de hacer la correlación ya lo tendremos creado y listo para su uso. Se podría correlar un experimento únicamente con este archivo ya que el correlador tomaría la configuración por defecto.

Es desde el archivo de configuración (.v2d) desde donde se pueden modificar los parámetros de configuración que tomará el correlador. Es específico de cada experimento y tiene que ser hecho a mano por el operador, aunque muchas veces se puede tomar como plantilla el mismo archivo de experimentos anteriores. Podemos encontrar una explicación detallada de este archivo así como de todos los parámetros configurables en la documentación de DiFX:

https://www.atnf.csiro.au/vlbi/dokuwiki/doku.php/difx/vex2difx

Con estos dos archivos se puede generar el archivo de control input (.input) utilizando la herramienta *vex2difx*.

6.4.2. Archivo de control (.input)

En este archivo es donde se puede encontrar la mayoría de información relativa a la configuración. Se genera con la herramienta *vex2difx* y es una combinación de los archivos V2D y VEX. La estructura se compone de diferentes tablas que contienen información sobre el experimento y parámetros que describen como se debe realizar la correlación, por ejemplo, relativas al experimento tenemos la fecha de inicio de este, su duración, el formato de los datos, etcétera.

6.4.3. Archivos de paralelización (.threads y .machines)

Anteriormente, en la sección de la estructura de DiFX, se ha explicado la capacidad de paralelización que tiene este correlador, pudiendo tener varios nodos enviado datos y varios nodos correlando al mismo tiempo. Esto se puede configurar, asignando los recursos de la mejor forma posible que considere el operador, editando los archivos .threads y .machines. No es algo necesario, pues DiFX es capaz de generar una configuración por defecto, pero no se ha diseñado para realizar esta función por lo que en la mayoría de casos es deficiente.

- **.machines**

  Es un archivo ASCII con N líneas, donde N es el número de procesos MPI que se van a generar. Cada línea contiene el nombre de un nodo, pudiendo repetirse todas las veces necesarias. En este caso, se lanzarán tantos procesos como línea con el nombre de dicho nodo. El orden en el que se escriben los nodos es importante, el primero corresponde al FxManager, el resto a los DataStreams con un número máximo igual al número de DataStreams necesarios (los datos de una antena pueden estar divididos en varias partes, necesitando más DataStreams), y los último a los Cores. Podemos ver un ejemplo en el que necesitemos cuatro DataStreams para la correlación:

```
alnilam    # FxManager
alnilam    # DataStream 1
alnilam    # DataStream 2
alnilam    # DataStream 3
alnilam    # DataStream 4
alnitak    # Core
mintaka    # Core
```

- **.threads**

  Este fichero también es ASCII y es igual de simple que .machines. La información que guarda se corresponde a la cantidad de hilos que van a generar los Cores. Siempre va a empezar por la misma línea que indica la cantidad de Cores que hay en el clúster. De forma similar que en el archivo anterior, cada línea establece el número de hilos de los Cores, en el mismo orden del que están escritos en .machines. Siguiendo el ejemplo anterior, supongamos que la CPU de alnitak es capaz de generar 8 hilos, mientras que la de mintaka puede generar 16. Entonces, el archivo .threads sería el siguiente.

```
NUMBER OF CORES:    2   # Num de Cores
8                       # Hilos de alnitak
16                      # Hilos de mintaka
```

  Es importante que se siga el formato mostrado, con una tabulación en la primera línea y con cada número en una línea.

6.5. Ficheros más relevantes

DiFX se compone de una gran cantidad de ficheros y librerías que hacen posible su funcionamiento. Los que interesan para este proyecto son los que componen al Core, en concreto los que se encargan de realizar la rotación de franja, la FFT y la corrección del retardo fraccionario.

- **Core.cpp**

  Lleva el control del Core realizando las tareas de coger los datos del buffer circular, organizar las subintegraciones y crear y lanzar los hilos. Al iniciarse se queda en un bucle esperando las instrucciones del FxManager de empezar a procesar datos. Una vez recibida

la instrucción sale del bucle para leer los datos, crear los hilos y lanzarlos con el fragmento de datos que le corresponde a cada uno. Si no se pueden lanzar más subintegraciones se queda esperando a que se termine alguna para volver a realizar todo el proceso descrito.

- **Mode.cpp**

  Es parte del Core y tiene todo lo necesario para procesar los datos del experimento. Aquí se hace la rotación de franja, la FFT y la corrección del retardo fraccionario. Cada hilo lanzado por el Core ejecuta una instancia de este código.

- **Architecture.h**

  Estructura las funciones matemáticas y las variables. Contiene macros que redefinen el nombre de las funciones para que sea común independientemente de la librería que se utilice, al igual que con el nombre de los tipos de datos. De esta manera se puede modificar la forma en la que se hacen las operaciones matemáticas únicamente cambiando este encabezado y a la hora de compilar el proyecto se elegirá una configuración u otra en función de lo que se haya indicado.

Capítulo 7

Planificación de las modificaciones

El objetivo del proyecto es mejorar el rendimiento del correlador DiFX utilizando una GPU para que las correlaciones logren completarse en menos tiempo del actual. En este apartado se va a a exponer el proceso seguido desde los primeros planteamientos expuestos, la elección de uno de ellos, sus razones y el diseño.

El primer paso fue entender la documentación de DiFX para identificar las partes que podían ser modificadas, que fue el Core. Después se buscaron los ficheros que se ejecutan en el Core: core.cpp, mode.cpp y architecture.h. Con los archivos identificados se procedió a la lectura del código viendo las librerías utilizadas y el modo de uso, así como la estructura del código, identificando qué partes del código realizaban las operaciones propias de un correlador DiFX (ver capítulo 6).

Con la lectura del código también se vieron las partes que podían ser modificadas para que hicieran uso de la GPU. Se identificaron dos tipos de operaciones: operaciones algebraicas (operaciones con vectores y matrices), y la FFT. Se analizó para ambas partes los cambios necesarios y su viabilidad de llevarlos a cabo. También se llevaron a cabo pruebas sobre cada caso. Estas pruebas se realizaron en el servidor alnitak, que donde está instalada la GPU.

7.1. Operaciones algebraicas

Estas operaciones son principalmente operaciones con vectores, como la suma de vectores, multiplicación, etcétera. Además hay otros tipos como establecer todos los elementos de un vector a cero o copiar un vector a otra variable. La librería Intel IPP se encarga de llevar a cabo todas estas operaciones aunque también existen funciones programadas a mano por los creadores de DiFX. Sin embargo, esta opción no se utiliza debido a que es más ineficiente que Intel IPP.

Como alternativa para la GPU se plantearon dos opciones. La primera era programar las funciones a mano con CUDA e implementarlas y la segunda opción era buscar librerías que hicieran las mismas operaciones o similares. La librería que se decidió probar es cuBLAS, que es una implementación de BLAS (Basic Linear Algebra Subprograms o Subprogramas Básicos de Álgebra Lineal). El único inconveniente es que apenas implementa funciones para operar con vectores, pero tampoco existe otra librería que tenga funciones dedicadas a esas operaciones. Una de las principales razones es que el mayor rendimiento de las GPUs se alcanza con operaciones matriciales, por lo que ese es el principal uso que se les da. Al final se decidió utilizar una

solución mixta, programando las funciones que no estaban en cuBLAS.

En total se programaron la suma y multiplicación de vectores y el producto escalar.

7.1.1. Suma

Se programaron a mano tanto la función de suma de vectores para la CPU como para la GPU. Esta operación consiste en una suma de dos vectores de números complejos elemento a elemento. Las partes relevantes del código se muestran a continuación:

Para la CPU el código es bastante sencillo. Consiste en un bucle que va recorriendo ambos vectores y sumando cada elemento. El resultado se almacena en uno de los dos vectores utilizados para la suma.

```
for (int i = 0; i < size; i++)
{
    v[i] = a[i] + v[i];
}
```

En la programación en GPU, el código que se va a ejecutar en esta se denomina kernel. Este código lo ejecutará cada hilo de forma paralela y con los datos que estén guardados en la memoria del device. Para saber qué elemento del vector debe sumar cada hilo se utiliza el id de este. Cada hilo tiene asignado un id único, desde 0 hasta el número máximo de hilos que se hayan lanzado. El código se divide en dos: por un lado el kernel y por otro una función que se ejecuta en el host (puede ser el main) y que se encargará de crear las variables del device, enviar los datos del host al device y de lanzar el kernel con los hilos necesarios.

Kernel:

```
__global__ void vectorMul (float* A, float* B, size_t size)
{
    // Calculamos el id del hilo
    int id = threadIdx.x + blockIdx.x * blockDim.x;

    // Comprobamos que el hilo que se esta ejecutando es menor que el tamanno
    de los vectores
    if (id < size)
    {
        // Realizamos la suma
        B[id] = A[id] + B[id];
    }
}
```

Main:

```
int main(int argc, char* argv[])
{
    // Obtenemos el tamanno de los vectores, pasado como argumento por el
    usuario
    long vector_size = strtol(argv[1], NULL, 10);

    // Inicializamos los vectores y les asignamos valores
    float * a = (float*) malloc(vector_size * sizeof(float));
    float * v = (float*) malloc(vector_size * sizeof(float));

    for (int i = 0; i < vector_size;  i++)
    {
        v[i] = i;
        a[i] = 1;
```

```
14      }
15
16      // ******** Aqui comienza la configuracion para la GPU ********
17
18      // Inicializamos las variables que se utilizaran en el device
19      float* d_A;
20      float* d_v;
21
22      // Reservamos memoria en el device
23      cudaMalloc((void**)&d_A, vector_size * sizeof(float));
24      cudaMalloc((void**)&d_v, vector_size * sizeof(float));
25
26      // Copiamos los vectores 'a' y 'v' a las variables 'd_A' y 'd_V', es decir,
         estamos pasado los vectores del host al device
27      cudaMemcpy(d_A, a, sizeof(float) * vector_size, cudaMemcpyHostToDevice);
28      cudaMemcpy(d_v, v, sizeof(float) * vector_size, cudaMemcpyHostToDevice);
29
30      /*  Inicializamos las variables que definiran la cantidad de hilos.
31       *  threadsPerBlock indica el numero de hilos por cada bloque (cada bloque
        representa un nucleo de la GPU)
32       *  blocksPerGrid indica el numero de bloques o nucleos que se van a
        ejecutar.
33       */
34    int threadsPerBlock = vector_size;
35    int blocksPerGrid = 1;
36
37      // Si la cantidad de hilos que necesitamos supera el maximo de hilos
        soportado por un bloque, hay que calcular la cantidad de bloques que
        necesitaremos
38    if (vector_size > 256)
39    {
40      threadsPerBlock = 256;
41      blocksPerGrid = (int)(vector_size + threadsPerBlock - 1/ threadsPerBlock);
42    }
43
44      // Lanzamos el kernel, indicando el numero de hilos y bloques que tiene que
         ejecutar, asi como los parametros
45      vectorMul<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_v, vector_size);
46
47      // Sincronizamos los hilos, es decir, el programa no continua hasta que
        todos los hilos hayan terminado
48      cudaDeviceSynchronize();
49
50      // Pasamos el vector resultado del device al host
51      cudaMemcpy(v, d_v, sizeof(float) * vector_size, cudaMemcpyDeviceToHost);
52
53      // Liberamos las variables utilizadas en el device
54      cudaFree(d_A);
55      cudaFree(d_v);
56  }
```

La estructura de la función main es común para todos los demás ficheros de prueba: declaración de variables, reserva de memoria en el device, paso de datos del host al device, lanzamiento del kernel, recuperar los datos y liberar la memoria. Lo único que cambia de forma relevante es el contenido del kernel.

Los resultados obtenidos muestran un mejor rendimiento por parte de la CPU. Es algo de esperar teniendo en cuenta que la dificultad de sumar vectores no es muy alta para el procesador y que el paso de datos del host al device representa la mayor parte del tiempo de ejecución de la GPU. se pueden ver en la figura 7.1. En el eje de la abscisas se muestra el tamaño de los

vectores y en el eje de ordenadas los ticks de reloj del ordenador medidos. Además, a los datos de este ultimo eje se le ha aplicado un logaritmo de dos que permita visualizar los datos de forma más sencilla.

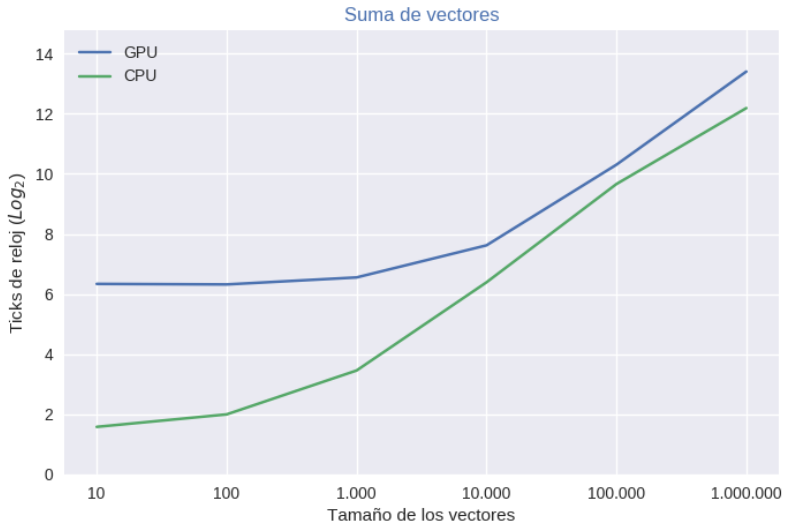


Figura 7.1: Suma de vectores

En la figura 7.2 se pueden ver los resultados de la misma prueba pero midiendo únicamente el tiempo de ejecución del kernel, obviando el tiempo que tarda en pasar los datos. Vemos que el rendimiento mejora aunque no llega a ser superior.

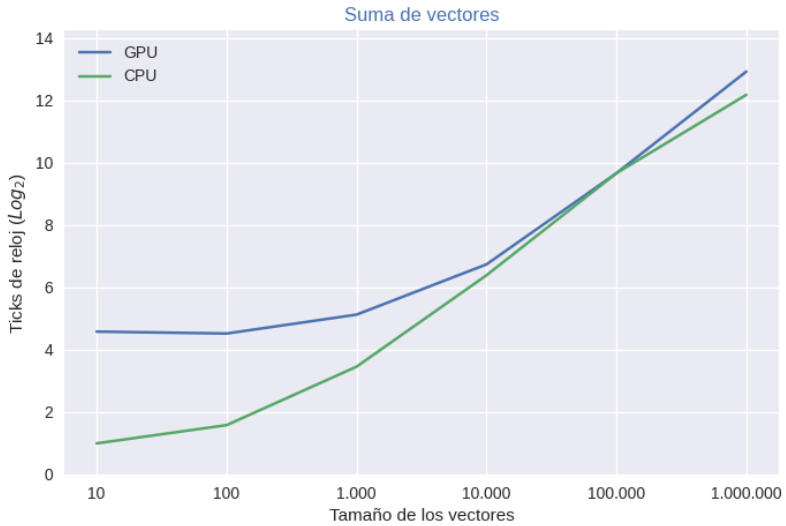


Figura 7.2: Suma de vectores. Para la GPU no se incluye el tiempo de envío de los datos entre CPU y GPU.

7.1.2. Multiplicación

Esta operación también se programó a mano, ya que no se encuentra en la librería cuBLAS. Consiste en multiplicar elemento a elemento ambos vectores, similar a la suma del apartado anterior. En la parte de GPU sólo se va a mostrar el código del kernel, debido a que la función main sigue la misma estructura que la de la suma.

El código que se ejecuta en la CPU consiste en un bucle que va recorriendo y multiplicando todos los elementos de los vectores y los almacena en uno de ellos.

```
for (int i = 0; i < size; i++)

v[i] = a[i] * v[i];
```

La parte del kernel también es bastante sencilla. Se obtiene el id del hilo y se realiza la operación de los elementos correspondientes al id de los vectores.

```
__global__ void vectorMul (float* A, float* B, size_t size)
{
  int id = threadIdx.x + blockIdx.x * blockDim.x;

  if (id < size)
      {
        B[id] = A[id] * B[id];
      }
}
```

Los resultados muestran que la GPU tiene mayor rendimiento que la CPU. En la figura 7.3 vemos la comparativa de ambos casos, incluyendo en la medida de la GPU el tiempo de pasar datos del host al device. A tamaños de datos más grandes la GPU consigue realizar la tarea más rápido, pero sin grandes diferencias.

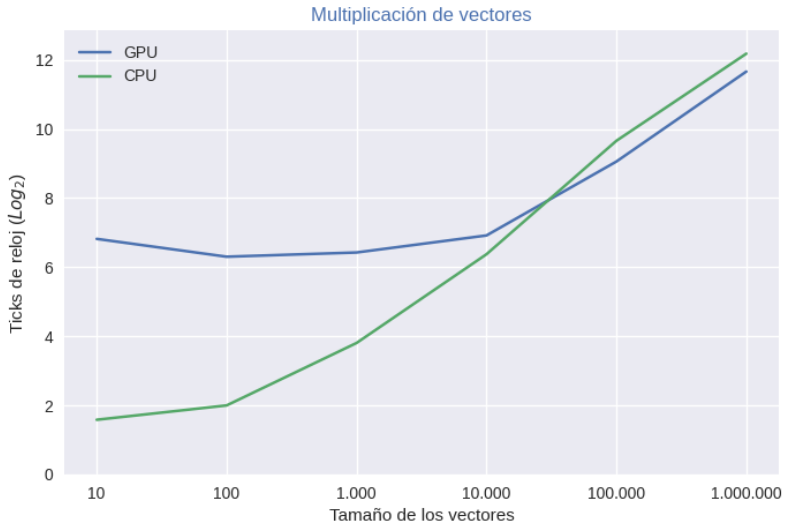


Figura 7.3: Multiplicación de vectores

Por otro lado tenemos las pruebas que no incluyen el tiempo de paso de los datos, en la figura 7.4. Aquí si podemos ver una gran diferencia entre el rendimiento de la GPU frente al de la CPU, siendo el primero mucho mejor.

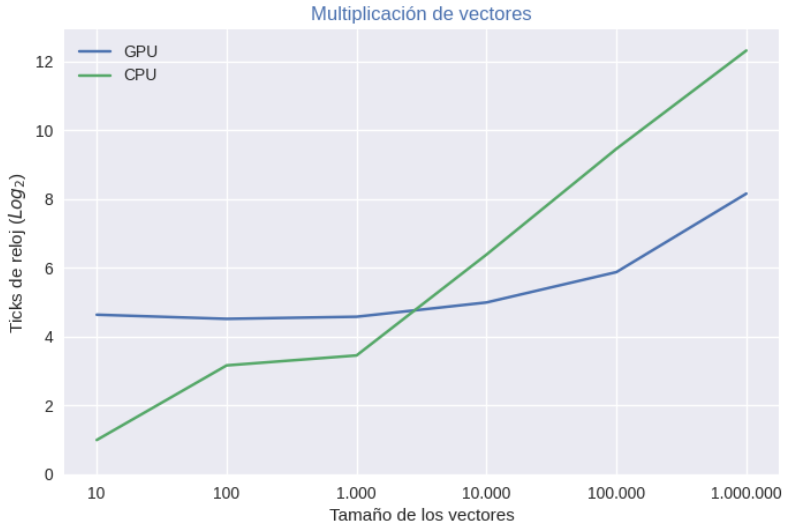


Figura 7.4: Multiplicación de vectores. Para la GPU no se incluye el tiempo de envío de los datos entre CPU y GPU. Se puede ver que existe un punto de intersección entre ambas funciones en, aproximadamente, una tamaño de vector de 5.000 elementos.

7.1.3. Producto escalar

Para el producto escalar se hizo uso de la librería cuBLAS, que proporciona una función para ello. El producto escalar se lleva a cabo según la ecuación

$$\sum_{i=1}^{n}(x[k] \times y[j]) \tag{7.1}$$

donde $k = 1 + (i-1) * incx$ y $j = 1 + (i-1) * incy$. Las variables $incx$ e $incy$ indican si hay que operar con todos los elementos de los vectores o de varios en varios elementos (por ejemplo, de dos en dos o de tres en tres). Para nuestro caso queremos que se tome todo el vector por lo que el valor de estas será 1. De esa forma se nos queda que $k = j = i - 1$, es decir, sigue la iteración de un bucle típico.

La operación de producto vectorial es bastante sencilla, un bucle va recorriendo ambos vectores a la vez que se multiplican los elementos y se va acumulando el resultado en una variable.

```
1 for (int i = 0; i < size; i++)
2 {
3     sum += a[i] * v[i];
4 }
```

En esta operación, al utilizar cuBLAS el código cambia respecto a los anteriores. El manejo de las operaciones dentro de la GPU las lleva a cabo la librería por lo que ya no tendremos un kernel ni tampoco habrá que preocuparse por la cantidad de hilos que se van a ejecutar. De lo que aún tendremos que hacernos cargo es de las variables de la GPU y asignarles memoria, así como de pasar los datos del host al device.

```
1 int main(int argc, char* argv[])
2 {
```

```
    // Obtenemos el tamanno de los vectores, pasado como argumento por el usuario
    int n = (int)strtol(argv[1], NULL, 10);

    // Declaramos el objeto de cuBLAS que planificara la operacion
    cublasHandle_t handle;

    // Declaramos los vectores y reservamos memoria
    float * a = (float*) malloc(n * sizeof(float));
    float * v = (float*) malloc(n * sizeof(float));
    float y;

    // Se rellenan los vectores
    for (int i = 0; i < n;  i++)
    {
            v[i] = i + 1;
    }

    for (int i = n; i < n + n; i++)
    {
            a[i - n] = i + 1;
    }

    // Inicializamos el objeto planificador de cuBLAS
    cublasCreate(&handle);
    cublasSetPointerMode(handle, CUBLAS_POINTER_MODE_DEVICE);

    // Declaramos los vectores que se utilizaran dentro de la GPU y les asignamos memoria
    float* d_A;
    float* d_v;
    float* d_y;
    cudaMalloc((void**)&d_A, n * sizeof(float));
    cudaMalloc((void**)&d_v, n * sizeof(float));
    cudaMalloc((void**)&d_y, sizeof(float));

    // Copiamos los vectores originales a las variables de la GPU
    cudaMemcpy(d_A, a, sizeof(float) * n, cudaMemcpyHostToDevice);
    cudaMemcpy(d_v, v, sizeof(float) * n, cudaMemcpyHostToDevice);

    // Llevamos a cabo la operacion
    cublasSdot(handle, n, d_A, 1, d_v, 1, d_y);

    // Esperamos a que acaben todos los hilos
    cudaDeviceSynchronize();

    // Obtenemos los resultados
    cudaMemcpy(&y, d_y, sizeof(float), cudaMemcpyDeviceToHost);

    // Liberamos las variables de la GPU
    cudaFree(d_A);
    cudaFree(d_v);
    cudaFree(d_y);

    return 0;
}
```

Los resultados obtenidos muestran una mayor diferencia entre el rendimiento de la GPU frente a la CPU de la que aparece en las anteriores pruebas.

En figura 7.5 la medida del tiempo también incluye el paso de memoria del host al device,

mientras que en la figura 7.6 no. En ambas la GPU obtiene mejores resultados a mayores tamaños de vectores, siendo mucho más clara en la segunda.

Es necesario comentar que se ha obviado el tiempo que lleva inicializar el objeto planificador de cuBLAS (*cublasHandle_t*). Este tiempo es bastante grande, llegando a una media de 300k. tiks de reloj, que sería 18.19 aplicado el logaritmo en base 2, siguiendo la escala utilizada en las gráficas. Esta cantidad (~ 18) es mucho mayor que el máximo que aparece en los resultados (~ 12). La razón por la que se ha obviado es que el planificador es reutilizable, por lo que únicamente sería necesario crearlo una vez y este tiempo de creación sería despreciable frente al tiempo total de la ejecución del programa. Más adelante se explicará el funcionamiento de los planificadores que utiliza CUDA en sus librerías.

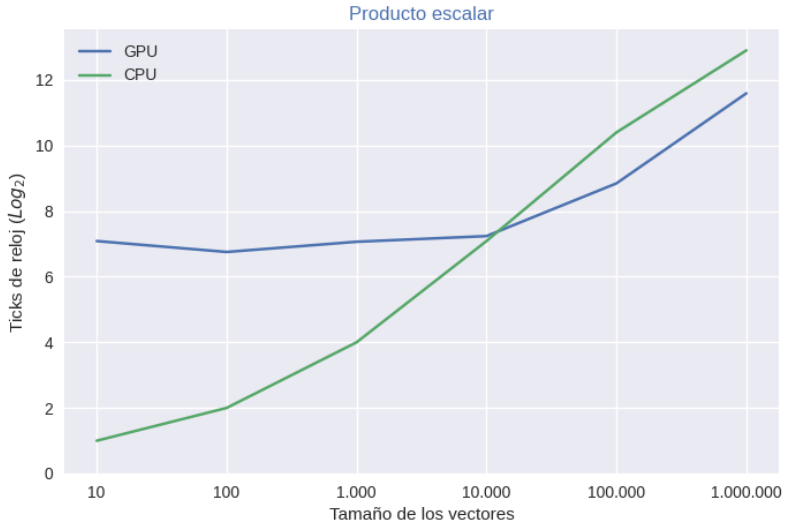


Figura 7.5: Producto escalar

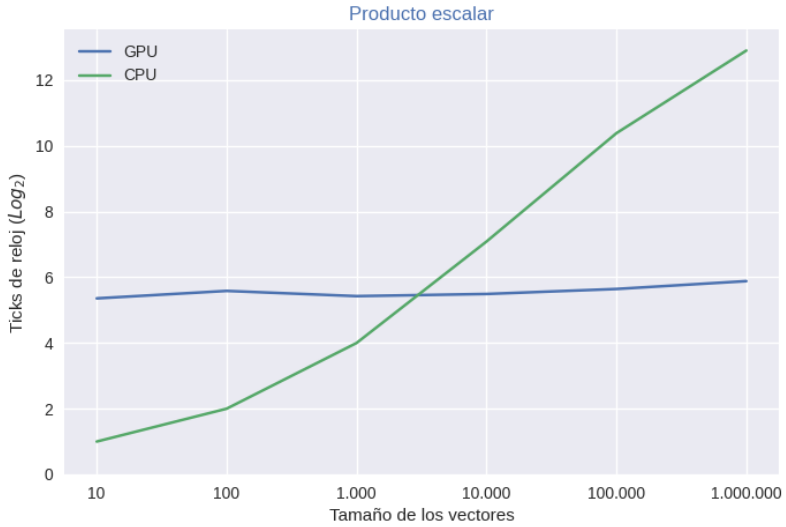


Figura 7.6: Producto escalar. Para la GPU no se incluye el tiempo de envío de los datos entre CPU y GPU.

7.1.4. Conclusiones

Los resultados de las pruebas muestran que la GPU es mejor en términos de tiempo de ejecución cuando se utilizan cantidades de datos grandes. Este punto de inflexión en el que la GPU pasa a ser mejor que la CPU depende de la operación a tratar. En la suma no se ha alcanzado. En la multiplicación ronda un tamaño de los vectores de 10.500 elementos contando el tiempo de pasar los datos o 1.500 elementos sin contar este tiempo. El producto escalar es algo similar, teniendo en cuenta el tiempo de traspaso, está en un tamaño de 10.000 elementos mientras que sin contar el tiempo está en 1.500 valores.

Esto era algo esperado, las tarjetas gráficas explotan al máximo la paralelización, por lo que a mayor cantidad de operaciones que pueda realizar a la vez mejores resultados se obtendrán frente a procesadores que realicen las operaciones de forma secuencial.

Como ya se ha ido mostrando, la medición del tiempo de ejecución se ha realizado de dos formas diferentes: teniendo en cuenta el tiempo de pasar los datos del host al device y sin tenerlo en cuenta. La razón de esto se debe a que lo ideal a la hora de realizar programas que utilicen la GPU como unidad de procesamiento para sus operaciones es mantener los datos en ella el mayor tiempo posible. Por ejemplo, si al resultado de realizar una multiplicación vectorial le voy a aplicar otra operación más adelante, podemos guardarla en la memoria global de la GPU para usarla más tarde.

En nuestro proyecto ese escenario sería el ideal a alcanzar. Sin embargo, la complejidad de realizarlo llevaría demasiado tiempo y esfuerzo. Hay que tener en cuenta que el correlador está diseñado y optimizado para trabajar sobre una CPU por lo que habría que cambiar parte de la estructura o incluso plantear una nueva.

Por otro lado, el tamaño de los vectores dentro del correlador no son demasiado grandes. Como referencia se ha utilizado el experimento *vo1021*, en el que se utiliza una resolución de 320 puntos por FFT, es decir que los vectores utilizados tendrán ese tamaño. Esta resolución es configurable pero no es recomendable que sea demasiado alta, ya que afecta a las FFT y entonces tenemos el problema de la variación de los retardos a lo largo de la ventana de integración.

La decisión final fue descartar esta idea y, en resumen, las condiciones que se han pensado para que se pudiese llevar a cabo es disponer de más tiempo y más recursos para plantearse una reestructuración de gran parte del código de DiFX.

7.2. FFT

La FFT es uno de los pasos fundamentales de DiFX, se realiza para transformar la señal entrante en el dominio del tiempo al dominio de la frecuencia. No hay una única función que lleve a cabo esta operación, sino que depende del tipo de datos que estemos tratando y la cantidad.

Respecto a la cantidad tenemos la *Transformada de Fourier Discreta* o *DFT* (*Discrete Fourier Transform*). Este algoritmo se utiliza para datos que tienen un tamaño que no es potencia de dos. Para los datos con un tamaño que es potencia de dos se utiliza la FFT. Aunque la DFT sea la más utilizada, en la mayoría de veces se utiliza el término FFT para referirse a la transformada de Fourier en DiFX.

Por otro lado, habrá que utilizar diferentes funciones dependiendo del tipo de datos de

entrada que tengamos y del tipo de datos de salida que deseemos. Como entrada podemos tener tanto datos reales como complejos, al igual que en la salida. Sin embargo, no valen todas las combinaciones. Si la entrada es compleja, la salida será compleja o real, y si la entrada es real siempre tendremos una salida compleja. Además, el tamaño de los datos de entrada y salida varían, como se ilustra en la tabla 7.1. Para el caso de entrada y salida compleja (C2C) tenemos la misma cantidad de datos de entrada como de salida. Sin embargo, para los casos de real a complejo (R2C) y de complejo a real (C2R), tenemos que los datos o dividen su tamaño entre dos o lo multiplican.

Tipo de FFT	Tipo de datos de entrada	Tipo de datos de salida
C2C	x números complejos	x números complejos
R2C	x números reales	$\lfloor x/2 \rfloor + 1$ números complejos
C2R	$\lfloor x/2 \rfloor + 1$ números complejos	x números reales

Tabla 7.1: Tamaño esperado de los datos de entrada y de salida

En DiFX únicamente se utilizan C2C y R2C, es decir, como salida siempre tendremos un vector de números complejos.

La librería de CUDA diseñada para la realización de la FFT es cuFFT y es la que implementaremos en el correlador. Originalmente, quien se encarga de esta tarea en DiFX es la librería Intel IPP, por lo que habrá que comparar ambas y ver sus similitudes y sus diferencias.

7.2.1. Comparativa entre cuFFT e Intel IPP

Tanto cuFFT como Intel IPP siguen los mismos pasos y utilizan funciones muy semejantes para llevar a cabo la FFT. Únicamente existen dos diferencias notables, que son los tipos de datos utilizados y la cantidad de datos aceptados por cada llamada a la función de ejecutar la FFT. También existen otras diferencias menos notables que se comentarán más resumidamente.

La primera diferencia que nos encontramos es en el tipo de datos que usa cada librería. En la tabla 7.2 se pueden ver algunos de los tipos originales de C++ y su correspondiente tipo en Intel IPP y cuFFT.

C++	Intel IPP	cuFFT
int	Ipp32s	-
float	Ipp32f	cufftReal
double	Ipp64f	cufftDoubleReal
complex float	Ipp32fc	cufftComplex
complex double	Ipp64fc	cufftDoubleComplex

Tabla 7.2: Comparativa de los tipos de datos de C++, Intel Ipp y cuFFT

Todos los tipos son semejantes, es decir, las variables complejas de IPP se componen de una parte variable real y una imaginaria, al igual que ocurre para cuFFT. Sin embargo, no es posible convertir una variable de tipo Ipp64fc (variable compleja de IPP) a una del tipo cufftComplex (variable compleja de cuFFT), como se haría en C++ para una conversión del tipo de una variable y que se puede ver a continuación:

```
1 int int_var = 2;
2 float f_var = (float) int_var;
```

Esto no presenta ningún problema serio, ya que se puede programar una función que haga el cambio de tipo:

```
inline int copyFromCudaToIntel(cf32_c* src, cf32* dst, int length)
{
  for (int i = 0; i < length; i++)
  {
    dst[i].re = src[i].x;
    dst[i].im = src[i].y;
  }
  return vecNoErr;
}
```

La otra diferencia importante es la cantidad de datos que acepta la función que ejecuta la FFT. Intel IPP la realiza de forma intuitiva. Para cada llamada a la función, se realiza la FFT de un vector con la cantidad de puntos deseada. Por otro lado, cuFFT es capaz de realizar más de una FFT por cada llamada. Para entenderlo mejor, vamos a referirnos a la cantidad de puntos de cada vector como NX y la cantidad de vectores como BATCH. Poniendo un ejemplo, si tenemos 4 vectores (BATCH de 4) con 8 puntos cada uno (NX de 8), IPP tendría que hacer 4 llamadas a la función con cada uno de los vectores, mientras que cuFFT únicamente realiza una llamada, introduciendo los cuatro vectores a la vez.

Los términos de NX y BATCH los introduce cuFFT. En cada llamada se pueden configurar estos valores según las necesidades del usuario. La salida de cuFFT será la misma estructura de datos, una cantidad de vectores igual al BATCH configurado y una cantidad de puntos en función del tipo de transformada que sea (ver tabla 7.1).

También hay que tener en cuenta que en la GPU se trabaja siempre con vectores, aunque la entrada sea una matriz (en estos casos se transforma la matriz en un vector, concatenando todas las filas o las columnas, según se requiera). Por lo que, usando IPP, lo que sería una matriz de la que se va haciendo la FFT de cada fila o cada matriz, con cuFFT tienes un solo vector que contiene todos los valores de la matriz.

Otras diferencias menos significativas son los parámetros que recibe cada función, así como la cantidad que tiene cada librería. Mientras que cuFFT utiliza una sola función para la FFT, Intel IPP utiliza dos: una para la DFT y otra para la FFT.

Respecto a las similitudes, ambos utilizan un planificador que hay que declarar y crear con anterioridad. La información que contiene es sobre el tamaño de los datos que se van a utilizar, el tipo de datos de entrada y salida que deseamos o el tipo de FFT, directa o inversa. También hay otros parámetros menos importantes que son más específicos de cada librería.

Analizando las similitudes y diferencias, la conclusión es que la portabilidad de Intel IPP a cuFFT no tiene gran complejidad. El principal problema surge en la estructuración de los datos previos así como del almacenamiento de los resultados. Si se trabaja con IPP, el proyecto va a estar diseñado para utilizar matrices, mientras que si se trabaja con cuFFT se diseñará para trabajar con vectores. Este problema implica que a la hora de implementar cuFFT, tengamos también que implementar un método que convierta las matrices en vectores y viceversa, añadiendo complejidad y tiempo de ejecución al proyecto.

7.2.2. Preparación de la pruebas de rendimiento entre Intel IPP y cuFFT

Antes de empezar a programar dentro del correlador es importante conocer bien ambas librerías. Para ello se han realizado varios programas con la intención de reproducir lo mejor

posible el escenario del DiFX. Además, también se pretendía conocer el comportamiento de ambas con diferentes tipos de datos y tamaños de vectores, así como de cantidad de vectores.

Primero veremos la parte de Intel IPP, explicando de forma detallada las partes más importantes del programa realizado. Después se hará lo mismo con cuFFT y, finalmente, se explicarán las conclusiones.

- **Intel IPP**

  Esta librería está desarrolla por Intel y diseñada para aprovechar al máximo sus procesadores. A parte de las funciones de la DFT y la FFT también hay funciones encargadas de reservar y liberar memoria, sustituyendo a las de C++ *malloc* y *delete*.

  La estructura seguida para realizar la transformada de Fourier se divide en dos partes: la preparación del plan y la obtención de los datos, y la propia función que lleve a cabo la DFT o la FFT, según corresponda.

  Los datos utilizados en las pruebas intentan ser una simulación de lo que sería una observación real, donde utilizan únicamente cuatro números. Los vectores de entrada se generan con estos cuatro valores, puestos de forma aleatoria, en un fichero aparte del programa principal que es capaz de devolver matrices tanto de números reales como de números complejos. En estas matrices, cada fila es un vector al que se le va a realizar la FFT, por lo que la cantidad de columnas corresponde con NX y la cantidad de filas con BATCH.

  A continuación se muestra la función que devuelve una matriz de números reales. Para el caso de los complejos, se utiliza un vector con el doble de tamaño y los valores del número complejos van por pares, es decir, los dos primeros valores corresponden a un número complejo (primero la parte real y después la imaginaria), los siguientes dos valores al segundo número, etcétera.

```
float** genMatrixReal(int NX, int BATCH, int seed)
    {
        // Valores utilizados en observaciones reales
        float values[] = {-3.533, -1.0, 1.0, 3.533};

        // Reservamos memoria para la matriz
        float **matrix = (float**) malloc(sizeof(float*) * NX * BATCH);

        // Establecemos la semilla para generar los numeros aleatorios
        srand(seed);

        // Rellenamos la matriz
        for (int i = 0; i < BATCH; i++)
        {
                // Reservamos memoria para cada fila de la matriz
                matrix[i] = (float*) malloc(sizeof(float) * NX);
                for (int j = 0; j < NX; j++)
                {
                        matrix[i][j] = values[rand() % 4];
                }
        }
        return matrix;
    }
```

  Una vez están los datos listos ya podemos empezar a preparar la FFT. IPP utiliza tres buffers de memoria, uno para guardar la estructura del plan y otros dos como espacio de

trabajo tanto para crear el plan como para realizar la FFT o la DFT. Para el caso de la FFT se necesita una variable extra, por lo que utiliza cuatro en vez de tres buffers.

Antes de crear el plan IPP necesita saber cuanta memoria tiene que reservar para estos buffers. Para ello utiliza una de las funciones de la tabla 7.3 en función de las necesidades. En el tipo de salida vemos que puede ser o un vector de números reales o un vector de números complejos. En cualquiera de los casos, la salida de la transformada es en números complejos, siendo los vectores de números reales una representación de estos números complejos en uno de los tres formatos utilizados por Intel: Pack, Perm o CCS, que se verán más adelante.

Nombre	Tipo de transformada	Tipo de entrada	Tipo de salida
ippsFFTGetSize_R_32f	FFT	Real	Vector de números reales
ippsFFTGetSize_C_32f	FFT	Compleja	Vector de números reales
ippsFFTGetSize_C_32fc	FFT	Compleja	Vector de números complejos
ippsDFTGetSize_R_32f	DFT	Real	Vector de números reales
ippsDFTGetSize_C_32f	DFT	Compleja	Vector de números reales
ippsDFTGetSize_C_32fc	DFT	Compleja	Vector de números complejos

Tabla 7.3: Funciones de Intel IPP para calcular la memoria necesaria de cada buffer

Los parámetros que recibe esta función dependen de si vamos a utilizar la FFT o la DFT. Aunque hay algunos comunes como los punteros a las variables que almacenan el tamaño necesario para los buffers, la bandera que indica si la salida de la transformada debe ser normalizada o no y una indicación que actualmente ya está en desuso. Además, a la DFT debemos indicarle la cantidad de puntos NX de los que se compone el vector de entrada y a la FFT el exponente n de la operación $NX = 2^n$, es decir, $n = log_2(NX)$.

Una vez ya sabemos el tamaño requerido para cada uno de los buffer podemos pasar a reservar la memoria.

El siguiente paso es crear el plan, para ello se van a utilizar las funciones de la tabla 7.4. Son similares a las anteriores, hay para FFT y para DFT, y los datos de entrada y salida siguen el mismo concepto.

Nombre	Tipo de transformada	Tipo de entrada	Tipo de salida
ippsFFTInit_R_32f	FFT	Real	Vector de números reales
ippsFFTInit_C_32f	FFT	Compleja	Vector de números reales
ippsFFTInit_C_32fc	FFT	Compleja	Vector de números complejos
ippsDFTInit_R_32f	DFT	Real	Vector de números reales
ippsDFTInit_C_32f	DFT	Compleja	Vector de números reales
ippsDFTInit_C_32fc	DFT	Compleja	Vector de números complejos

Tabla 7.4: Funciones de Intel IPP para crear el plan de la transformada de Fourier

Los parámetros que recibe también dependen de si se va a realizar una FFT o una DFT. Los comunes siguen siendo la bandera y la indicación, que aunque ya se habían indicado en *ippsGetSize* hay que volver a ponerlo. Para la FFT tenemos un puntero a la variable que almacenará el plan y dos de los tres buffers, uno con la memoria donde se creará la estructura del plan y otro como zona de trabajo, y el exponente n que habíamos calculado

antes. Para la DFT tenemos uno de los buffers como zona de trabajo, la variable que almacenará el plan y la cantidad de puntos NX del vector de entrada.

En este punto ya está disponible todo lo necesario para llevar a cabo la transformada de Fourier. Tenemos una gran cantidad de funciones que responden a las diferentes necesidades del usuario. En primer lugar podemos distinguir entre transformada directa o inversa, en este proyecto únicamente nos interesan las transformadas directas. Después las podemos clasificar en función del formato en que va a devolver los resultados, que son Pack, Perm y CCS. En DiFX solo se utiliza CCS por lo que es el único que se tendrá en cuenta. Por último, tenemos tantas funciones como en los otros apartados, dependiendo del formato de datos de entrada y de salida. En la tabla 7.5 podemos ver las funciones que más nos interesan.

Nombre	Tipo de transformada	Tipo de entrada	Tipo de salida
ippsFFTFwd_RtoCCS_32f	FFT	Real	CCS
ippsFFTFwd_CtoC_32f	FFT	Compleja	Compleja
ippsFFTFwd_CtoC_32fc	FFT	Compleja	Compleja
ippsDFTFwd_RtoCCS_32f	DFT	Real	CCS
ippsDFTFwd_CtoC_32f	DFT	Compleja	Compleja
ippsDFTwd_CtoC_32fc	DFT	Compleja	Compleja

Tabla 7.5: Funciones de Intel IPP para calcular la transformada de Fourier de un vector dado

Por último antes de pasar a ver el código, se va a explicar por encima el formato CCS. Es bastante simple y consiste en almacenar los resultados de la transformada con números reales. No se guardan de la forma sencilla de poner parte real e imaginaria juntas e ir concatenando números. Se sigue otra lógica similar que se puede ver en la tabla 7.6.

Índice	0	1	2	3	...	N-2	N-1	N	N+1
CCS - longitud par	R_0	0	R_1	I_1	...	$R_{N/2-1}$	$I_{N/2-1}$	$R_{N/2}$	0
CCS - longitud impar	R_0	0	R_1	I_1	...	$I_{(N-1)/2-1}$	$R_{(N-1)/2}$	$I_{(N-1)/2}$	-

Tabla 7.6: Formato CCS

Finalmente, tenemos el código utilizado para llevar a cabo las pruebas. Se ha documentado para que se pueda realizar un fácil seguimiento, tomando como guía la explicación anterior.

Funciones para la FFT y DFT: Se han separado las funciones que realizan la transformada ya que se quiso seguir el concepto utilizado en DiFX. Sin embargo, esto no era estrictamente necesario.

```
1  void vectorDFT_RtoC(float *Src, Ipp32fc *Dst, IppsDFTSpec_R_32f *pSpec,
       Ipp8u *pMemBuffer)
2  {
3          /// DFT directa real a CCS
4          ippsDFTFwd_RToCCS_32f((Ipp32f*) Src, (Ipp32f*) Dst, pSpec,
       pMemBuffer);
5  }
6
7
8  void vectorDFT_CtoC(Ipp32fc *Src, Ipp32fc *Dst, IppsDFTSpec_C_32fc *pSpec,
       Ipp8u *pMemBuffer)
9  {
10         /// DFT directa complejo a complejo
```

```
11          ippsDFTFwd_CToC_32fc(Src, Dst, pSpec, pMemBuffer);
12  }
13
14
15  void vectorFFT_RToC(float *Src, Ipp32fc *Dst, IppsFFTSpec_R_32f *pSpec,
        Ipp8u *pMemBuffer)
16  {
17          /// FFT directa real a CCS
18          ippsFFTFwd_RToCCS_32f(Src, (Ipp32f*) Dst, pSpec, pMemBuffer);
19
20  }
21
22
23  void vectorFFT_CToC(Ipp32fc *Src, Ipp32fc *Dst, IppsFFTSpec_C_32fc *pSpec,
        Ipp8u *pMemBuffer)
24  {
25          /// FFT directa compleja a compleja
26          ippsFFTFwd_CToC_32fc(Src, Dst, pSpec, pMemBuffer);
27  }
```

Función principal: Aquí va lo comentado en el proceso de realización de una transformada de Fourier con Intel IPP. Hay cuatro partes: primero se divide dependiendo si la entrada es compleja o real, y luego por si hay que hacer FFT o DFT. Estas cuatro partes de código son bastante semejantes pero, debido a que utilizan diferentes variables, tipos de datos y funciones se ha decidido hacerlas de forma separada para una mejor comprensión a la hora de leerlo.

```
1   int main(int argc, char *argv[])
2   {
3       // Variables auxiliares
4       int cmplx = 1;
5       int init = 1;
6
7       // Comprobamos los parametros introducidos por el usuario
8       if (!strcmp(argv[1], "-r"))
9     {
10          cmplx = 0;
11          init = 2;
12    }
13
14      // Almacenamos los valores introducidos por el usuario
15      int NX = strtol(argv[init], NULL, 10);
16      int BATCH = strtol(argv[init + 1], NULL, 10);
17
18      // Semilla para generar los numeros aleatorios
19    int SEED = 1;
20
21      // Bandera que indica que no se realice ninguna normalizacion de los
        resultados de las transforadas
22      int flag = IPP_FFT_NODIV_BY_ANY;
23
24      // Variables para guardar el tamanno de los buffer
25      int sizeSpec= 0;
26      int sizeInit = 0;
27      int sizeBuffer = 0;
28
29      // Buffers
30      Ipp8u *pMemSpec = 0;
31      Ipp8u *pMemInit = 0;
32      Ipp8u *pMemBuffer = 0;
33
34      // Variables para medir el tiempo de ejecucion de la parte que nos
```

```
    interese
35  clock_t start, end;
36
37  // Si se ha elegido la opcion de entrada compleja
38  if (cmplx)
39  {
40      printf("*************CPU Complex*************\n");
41      printf("Size: %d\n  NX:  %d\n  BATCH: %d\n\n", NX * BATCH, NX,
    BATCH);
42
43      // Declaramos la variable que guardara los datos
44      float **datos_ = (float**) malloc(sizeof(float*) * NX * BATCH);
45
46      // Obtenemos los datos complejos. Estos vienen en un vector de
    numeros reales
47      datos_ = genMatrixComplex(NX, BATCH, SEED);
48
49      // Pasamos los datos a una variable Ipp32fc**. Cada par del vector
    datos_ corresponde a un numero complejo
50      Ipp32fc **datos = (Ipp32fc**) malloc(sizeof(Ipp32f*) * NX * BATCH);
51      for (int i = 0; i < BATCH; i++)
52      {
53          datos[i] = (Ipp32fc*) malloc(sizeof(Ipp32f) * NX * 2);
54          for (int j = 0; j < NX * 2; j += 2)
55          {
56              datos[i][j / 2].re = datos_[i][j];
57              datos[i][j / 2].im = datos_[i][j + 1];
58          }
59      }
60
61      // Variable que almacenara los resultados
62      Ipp32fc **dst = (Ipp32fc**) malloc(sizeof(Ipp32fc*) * NX * BATCH);
63
64      // Comprobamos que NX sea potencia de dos
65      if (!((NX) & (NX - 1)))
66      {
67          // Si es potencia de dos entonces utilizamos la FFT
68          printf("**Using FFT**\n\n");
69
70          // Variable que guardara el plan de la FFT
71          IppsFFTSpec_C_32fc *pSpec = 0;
72          int order = log2 (NX);
73
74          // Obtememos el tamanno necesario para los buffers
75          ippsFFTGetSize_C_32fc(order, flag, ippAlgHintNone, &sizeSpec, &
    sizeInit, &sizeBuffer);
76
77          /* Reservamos la memoria necesaria para los buffers.
78           *  pMemSpec es la zona donde se guarda la estructura de la FFT
79           *  pMemInit es la zona de trabajo de la funcion que crea el
    plan
80           *  pMemBuffer es la zona de trabajo de la FFT
81          */
82          pMemSpec = (Ipp8u*) ippMalloc(sizeSpec);
83
84          if (sizeInit > 0)
85          {
86              pMemInit = (Ipp8u*) ippMalloc(sizeInit);
87          }
88
89          if (sizeBuffer > 0)
90          {
91              pMemBuffer = (Ipp8u*) ippMalloc(sizeBuffer);
```

```
92              }
93
94              // Creamos el plan para la FFT
95              ippsFFTInit_C_32fc(&pSpec, order, flag, ippAlgHintNone,
    pMemSpec, pMemInit);
96
97              // Liberamos la memoria del buffer que no vamos a volver a
    utilizar
98              if (sizeInit > 0)
99              {
100                 ippFree(pMemInit);
101             }
102
103             /* Comenzamos a grabar el tiempo.
104              * Se ha elegido este punto debido a que todo lo relacionado
    con el plan solo se necesita hacer una vez. Ya
105              * que el objeto puede ser reutilizado siempre que tengamos el
    mismo tipo de datos de entrada y la misma cantidad.
106        */
107             start = clock();
108
109       /* Hacemos la FFT directa de complejo a complejo.
110              * Utilizamos un bucle que hace un numero de BATCH iteraciones,
     es decir, recorre todos los vectores.
111              * Para cada vector se hace una FFT
112             */
113             for (int i = 0; i < BATCH; i++)
114             {
115                 dst[i] = (Ipp32fc*) malloc(sizeof(Ipp32fc) * NX);
116                 vectorFFT_CToC(datos[i], dst[i],  pSpec, pMemBuffer);
117             }
118
119             // Paramos de grabar el tiempo
120             end = clock();
121         }
122         else
123         {
124             // Si no es potencia de dos entonces utilizamos la DFT
125             printf("**Using DFT**\n\n");
126
127             // Variable que guardara el plan de la DFT
128             IppsDFTSpec_C_32fc *pSpec = 0;
129
130             // Obtememos el tamanno necesario para los buffers
131             ippsDFTGetSize_C_32fc(NX, flag, ippAlgHintNone, &sizeSpec, &
    sizeInit, &sizeBuffer);
132
133       /* Reservamos la memoria necesaria para los buffers.
134              *  pMemSpec es la zona donde se guarda la estructura de la FFT
135              *  pMemInit es la zona de trabajo de la funcion que crea el
    plan
136              *  pMemBuffer es la zona de trabajo de la FFT
137             */
138             pSpec = (IppsDFTSpec_C_32fc*)ippMalloc(sizeSpec);
139
140             if (sizeInit > 0)
141             {
142                 pMemInit = (Ipp8u*) ippMalloc(sizeInit);
143             }
144
145             if (sizeBuffer > 0)
146             {
147                 pMemBuffer = (Ipp8u*) ippMalloc(sizeBuffer);
```

```
148             }
149
150             // Creamos el plan para la DFT
151             ippsDFTInit_C_32fc(NX, flag, ippAlgHintNone, pSpec, pMemInit);
152
153             if (sizeInit > 0)
154             {
155                 ippFree(pMemInit);
156             }
157
158             /* Comenzamos a grabar el tiempo.
159              * Se ha elegido este punto debido a que todo lo relacionado
    con el plan solo se necesita hacer una vez. Ya
160              * que el objeto puede ser reutilizado siempre que tengamos el
    mismo tipo de datos de entrada y la misma cantidad.
161             */
162             start = clock();
163
164       /* Hacemos la FFT directa de complejo a complejo.
165              * Utilizamos un bucle que hace un numero de BATCH iteraciones,
     es decir, recorre todos los vectores.
166              * Para cada vector se hace una FFT
167             */
168             for (int i = 0; i < BATCH; i++)
169             {
170                 dst[i] = (Ipp32fc*) malloc(sizeof(Ipp32fc) * (NX));
171                 vectorDFT_CtoC(datos[i], dst[i],  pSpec, pMemBuffer);
172             }
173
174             // Paramos de grabar el tiempo
175             end = clock();
176         }
177
178         // Liberamos los buffer que hemos utilizado
179         if (sizeBuffer > 0)
180         {
181             ippFree(pMemBuffer);
182         }
183
184         ippFree(pMemSpec);
185     }
186     else
187     {
188         printf("*************CPU Real*************\n");
189         printf("Size: %d\n  NX:  %d\n  BATCH: %d\n\n", NX * BATCH, NX,
    BATCH);
190
191         // Declaramos la variable que guardara los datos
192         float **datos = (float**) malloc(sizeof(float*) * NX * BATCH);
193
194         // Obtenemos los datos reales
195         datos = genMatrixReal(NX, BATCH, SEED);
196
197         // Variable que almacena los resultados
198         Ipp32fc **dst = (Ipp32fc**) malloc(sizeof(Ipp32fc*) * (NX / 2 + 1)
    * BATCH);
199
200         // Comprobamos que NX sea potencia de dos
201         if (!((NX) & (NX - 1)))
202     {
203             // Si es potencia de dos entonces utilizamos la FFT
204             printf("**Using FFT**\n\n");
205
```

```
206             // Variable que guardara el plan de la FFT
207             IppsFFTSpec_R_32f *pSpec = 0;
208             int order = log2 (NX);
209 
210             // Obtememos el tamanno necesario para los buffers
211             ippsFFTGetSize_R_32f(order, flag, ippAlgHintNone, &sizeSpec, &
    sizeInit, &sizeBuffer);
212 
213             /* Reservamos la memoria necesaria para los buffers.
214              *  pMemSpec es la zona donde se guarda la estructura de la FFT
215              *  pMemInit es la zona de trabajo de la funcion que crea el
    plan
216              *  pMemBuffer es la zona de trabajo de la FFT
217             */
218             pMemSpec = (Ipp8u*) ippMalloc(sizeSpec);
219 
220             if (sizeInit > 0)
221             {
222                 pMemInit = (Ipp8u*) ippMalloc(sizeInit);
223             }
224 
225             if (sizeBuffer > 0)
226             {
227                 pMemBuffer = (Ipp8u*) ippMalloc(sizeBuffer);
228             }
229 
230             // Creamos el plan para la FFT
231             ippsFFTInit_R_32f(&pSpec, order, flag, ippAlgHintNone, pMemSpec
    , pMemInit);
232 
233             // Liberamos la memoria del buffer que no vamos a volver a
    utilizar
234             if (sizeInit > 0)
235             {
236                 ippFree(pMemInit);
237             }
238 
239             /* Comenzamos a grabar el tiempo.
240              * Se ha elegido este punto debido a que todo lo relacionado
    con el plan solo se necesita hacer una vez. Ya
241              * que el objeto puede ser reutilizado siempre que tengamos el
    mismo tipo de datos de entrada y la misma cantidad.
242             */
243             start = clock();
244 
245             /* Hacemos la FFT directa de complejo a complejo.
246              * Utilizamos un bucle que hace un numero de BATCH iteraciones,
     es decir, recorre todos los vectores.
247              * Para cada vector se hace una FFT
248             */
249             for (int i = 0; i < BATCH; i++)
250             {
251                 dst[i] = (Ipp32fc*) malloc(sizeof(Ipp32fc) * (NX / 2 + 1));
252                 vectorFFT_RToC(datos[i], dst[i],  pSpec, pMemBuffer);
253             }
254 
255             // Paramos de grabar el tiempo
256             end = clock();
257         }
258         else
259     {
260             // Si no es potencia de dos entonces utilizamos la DFT
261             printf("**Using DFT**\n\n");
```

```
262
263             // Variable que guardara el plan de la DFT
264             IppsDFTSpec_R_32f *pSpec = 0;
265
266             // Obtememos el tamanno necesario para los buffers
267             ippsDFTGetSize_R_32f(NX, flag, ippAlgHintNone, &sizeSpec, &
    sizeInit, &sizeBuffer);
268
269             /* Reservamos la memoria necesaria para los buffers.
270              *  pMemSpec es la zona donde se guarda la estructura de la FFT
271              *  pMemInit es la zona de trabajo de la funcion que crea el
    plan
272              *  pMemBuffer es la zona de trabajo de la FFT
273             */
274             pSpec = (IppsDFTSpec_R_32f*)ippMalloc(sizeSpec);
275
276             if (sizeInit > 0)
277             {
278                 pMemInit = (Ipp8u*) ippMalloc(sizeInit);
279             }
280
281             if (sizeBuffer > 0)
282             {
283                 pMemBuffer = (Ipp8u*) ippMalloc(sizeBuffer);
284             }
285
286             // Creamos el plan para la DFT
287             ippsDFTInit_R_32f(NX, flag, ippAlgHintNone, pSpec, pMemInit);
288
289             if (sizeInit > 0)
290             {
291                 ippFree(pMemInit);
292             }
293
294             /* Comenzamos a grabar el tiempo.
295              * Se ha elegido este punto debido a que todo lo relacionado
    con el plan solo se necesita hacer una vez. Ya
296              * que el objeto puede ser reutilizado siempre que tengamos el
    mismo tipo de datos de entrada y la misma cantidad.
297             */
298             start = clock();
299
300             /* Hacemos la FFT directa de complejo a complejo.
301              * Utilizamos un bucle que hace un numero de BATCH iteraciones,
     es decir, recorre todos los vectores.
302              * Para cada vector se hace una FFT
303             */
304             for (int i = 0; i < BATCH; i++)
305             {
306                 dst[i] = (Ipp32fc*) malloc(sizeof(Ipp32fc) * (NX / 2 + 1));
307                 vectorDFT_RtoC(datos[i], dst[i],  pSpec, pMemBuffer);
308             }
309             end = clock();
310         }
311
312         // Liberamos los buffer que hemos utilizado
313         if (sizeBuffer > 0)
314         {
315             ippFree(pMemBuffer);
316         }
317
318         ippFree(pMemSpec);
319
```

```
320         // Calculamos la duracion medida y la imprimimos
321         clock_t total = end - start;
322         printf("Total CPU: %d\n\n", total);
323 
324     }
325     return 0;
326 }
```

El código genera una salida por pantalla que puede ser redirigida a un fichero y de esta manera guardar los resultados. También se hizo que se guardase directamente en un fichero, para poder automatizar las pruebas planificadas, aunque ese código no se ha mostrado, dejando únicamente la impresión por pantalla, debido a que no es relevante y puede llegar a confundir.

Un ejemplo de la salida que genera por pantalla es el siguiente. En este caso tenemos una matriz con 64 filas, cada una con 320 puntos, y el tipo de números que tomará como entrada son complejos, indicado en la primera fila. Como ya hemos visto, para cada fila se hará una FFT. Más abajo se muestra el algoritmo utilizado, que pueden ser FFT o DFT. La cantidad de puntos que tenemos es 320, que no es potencia de dos, por lo que se ha realizado una DFT. Finalmente, podemos ver el tiempo total que ha tardado en realizarlas, indicando si es CPU o GPU. Esta indicación se utilizará más tarde para poder diferenciar la procedencia de los datos a la hora de pintar las gráficas.

```
*************CPU Complex*************
Size: 20480
  NX:  320
  BATCH: 64

**Using DFT**

Total CPU: 152
```

- **cuFFT**

Esta librería está desarrollada por NVIDIA para llevar a cabo la FFT en GPUs. Esta librería funciona en cualquier tarjeta que sea capaz de utilizar CUDA.

Al igual que Intel IPP, está altamente optimizada para aprovechar al máximo los recursos de las tarjetas. El rendimiento del algoritmo depende principalmente del tamaño de los datos de entrada. Este viene definido por $2^a \times 3^b \times 5^c \times 7^d$, siendo mejor contra más bajo sea el factor. El óptimo está, al igual que en la librería de Intel, con un factor de 2^a.

La datos utilizados como entrada se generan de la misma forma que en las pruebas de Intel IPP, intentando simular una entrada de una observación real.

Una vez tenemos los datos debemos pasarlos del host al device. La GPU actúa como una máquina aparte del resto del ordenador, con memoria propia, que se comunica con la CPU. Esto quiere decir que necesita variables propias que se almacenen en su memoria. El manejo de estas variables es controlado y gestionado por CUDA, así como el flujo de datos entre host y device. Existen una gran número de estas funciones pero para este proyecto solo se ha hecho uso de las siguientes:

- cudaMalloc: se encarga de reservar memoria en el device, similar a malloc de C++. Aunque con la diferencia de que no devuelve un puntero sino que se pasa como parámetro.

- cudaFree: libera memoria reservada en el device, similar a free de C++.

- cudaMemcpy: copia los datos de una variable del host a otra del device o viceversa. Consideramos una variable del host la que tiene memoria reservada en la memoria del ordenador y variable del device la que tiene memoria reservada en la memoria de la GPU.

- cudaDeviceSynchronize: sincroniza todos los hilos de la GPU.

El proceso general que se sigue para ejecutar un programa en la GPU pasa por reservar la memoria necesaria, pasar los datos al device, lanzar el código a ejecutar, sincronizar todos los hilos si es necesario, pasar los datos necesarios del device al host y liberar la memoria.

Algunos de estos pasos se pueden obviar dependiendo de la forma en la que se haya planteado la solución del problema que se quiere resolver. Es bastante común que los datos residan en la memoria del device el mayor tiempo posible, pudiendo evitar el paso de datos entre los dispositivos o reutilizando variables.

Con cuFFT este proceso también se debe seguir, añadiendo únicamente, para este proyecto, una función adicional: *cufftPlan1d()*. Esta será la encargada de crear el plan en función de la cantidad de Transformadas de Fourier que se vayan a realizar, el número de puntos de cada una y el tipo de FFT. El tipo de FFT hace referencia al tipo de datos de entrada y al tipo de datos de salida, pudiendo tener de datos reales a complejos o de datos complejos a reales, tanto de precisión simple (float) como de doble precisión (double). Además, existe una función para crear planes diferentes en función de las dimensiones de la entrada. En nuestro caso vamos a utilizar las de una dimensión ya que la entrada será una señal de radio en una dimensión. La creación del plan se puede realizar en cualquier momento antes de ejecutar la FFT.

Las funciones para llevar a cabo la Transformada de Fourier únicamente se diferencian por el tipo de datos de entrada y de salida, siendo independiente el tamaño de estos. Es por ello que no se diferencian entre DFT y FFT. Respecto a la salida de la FFT, esta es en vectores de números del tipo complejo de cuFFT (cufftComplex) o de números del tipo real de cuFFT (cufftReal).Podemos verlas en la tabla 7.7.

Nombre	Tipo de entrada	Tipo de salida
cufftExecC2C	Compleja	Compleja
cufftExecR2C	Real	Compleja
cufftExecC2R	Compleja	Real

Tabla 7.7: Funciones de cuFFT para ejecutar la Transformada de Fourier

Por último tenemos el código empleado para las pruebas de cuFFT. La estructura es muy similar a la utilizada para Intel IPP con algunos cambios. En vez de dividirse en cuatro partes, se divide únicamente en dos dependiendo de si la entrada es de números reales o complejos. Esto se debe a que no hace falta diferenciar entre DFT y FFT.

Respecto a la forma de imprimir por pantalla los resultados de la prueba se sigue el mismo

procedimiento. Respecto a las funciones que ejecutan la FFT, también se han separado de la función principal.

Funciones para la FFT: Estas funciones realizan más operaciones aparte de la Transformada de Fourier. También se encargan de pasar los datos del device al host y de convertirlos de un único vector a una matriz. Cada fila de esta matriz será el resultado de cada FFT realizada. El valor que se retorna es la marca temporal final que se utilizará para medir el tiempo de ejecución de la parte que nos interese.

```
void createPlan(cufftHandle *plan, int NX, int BATCH, char type)
{
    // Crea un plan para la FFT
    if (type == 'R')
    {
        // Plan para FFT con entrada real y salida compleja
        cufftPlan1d(plan, NX, CUFFT_R2C, BATCH);
    }
    else
    {
        // Plan para FFT con entrada compleja y salida compleja
        cufftPlan1d(plan, NX, CUFFT_C2C, BATCH);
    }
}


clock_t vectorFFT_R2C(cufftComplex **odata, cufftHandle plan, int NX, int
    BATCH, float *d_idata, cufftComplex *d_data, cufftComplex *odata_)
{
    // Lanzamos la FFT con entrada real y saida compleja
    cufftExecR2C(plan, d_idata, d_data);

    // Sincronizamos todos los hilos de la GPU
    cudaDeviceSynchronize();

    // Retornamos los datos al host. Debido a que la entrada es real, los
    vectores resultantes tendran una cantidad de puntos (NX / 2 + 1), y
    sera la cantidad de datos que se copien.
    cudaMemcpy(odata_, d_data, sizeof(cufftComplex) * (NX / 2 + 1) * BATCH,
     cudaMemcpyDeviceToHost);

    // Marca temporal final
    clock_t end = clock();

    // Pasamos el vector resultante a una matriz
        for (int i = 0; i < BATCH; i++)
        {
                odata[i] = (cufftComplex*) malloc (sizeof(cufftComplex) * (
    NX / 2 + 1));
                for (int j = 0; j < (NX / 2 + 1); j++)
                {
                        odata[i][j].x = odata_[j + i * (NX / 2 + 1)].x;
                        odata[i][j].y = odata_[j + i * (NX / 2 + 1)].y;
                }
        }
  return end;
}


clock_t vectorFFT_C2C(cufftComplex **odata, cufftHandle plan, int NX, int
    BATCH, cufftComplex *d_idata, cufftComplex *d_data, cufftComplex *
    odata_)
{
```

```
47      // Lanzamos la FFT con entrada compleja y salida complejas
48      cufftExecC2C(plan, d_idata, d_data, CUFFT_FORWARD);
49
50      // Sincronizamos todos los hilos de la GPU
51      cudaDeviceSynchronize();
52
53      // Retornamos los datos al host
54      cudaMemcpy(odata_, d_data, sizeof(cufftComplex) * NX * BATCH,
        cudaMemcpyDeviceToHost);
55
56      // Marca temporal final
57      clock_t end = clock();
58
59      // Pasamos el vector resultante a una matriz
60          for (int i = 0; i < BATCH; i++)
61          {
62                  odata[i] = (cufftComplex*) malloc (sizeof(cufftComplex) *
        NX);
63                  for (int j = 0; j < NX; j++)
64                  {
65                          odata[i][j].x = odata_[j + i * NX].x;
66                          odata[i][j].y = odata_[j + i * NX].y;
67                  }
68          }
69
70    return end;
71  }
```

Función principal: Las funciones generales de gestión de memoria de la GPU con CUDA se realizan en esta función. También se llevan a cabo de la parte de obtención de los datos, similar a como se hace en el código de las pruebas de Intel IPP, e interpretación de las opciones introducidas por el usuario.

```
1   int main(int argc, char *argv[])
2   {
3       // La primera operacion que lanza un proceso en la GPU consume una gran
         cantidad de tiempo, por temas de configuracion, que no queremos medir.
         Es por ello que realizamos una operacion trivial al comienzo.
4       int *p;
5       cudaMalloc((void**)&p, 1);
6       cudaFree(p);
7
8       int cmplx = 1;
9       int init = 1;
10
11      // Comprobamos los parametros introducidos por el usuario
12      if (!strcmp(argv[1], "-r"))
13      {
14          cmplx = 0;
15          init = 2;
16      }
17
18      // Variables para medir el tiempo de ejecucion
19      clock_t start, end;
20
21      /* NX    -> cantidad de puntos de un unico vector
22       * BATCH -> cantidad de vectores
23       * SEED  -> semilla para generar los numeros aleatorios
24      */
25      int NX = strtol(argv[init], NULL, 10);
26      int BATCH = strtol(argv[init + 1], NULL, 10);
27      int SEED = 1;
```

```
28      int size = NX * BATCH;
29
30      // Variable donde se almacenara el plan creado
31      cufftHandle plan;
32
33      // Comprobamos si la entrada que debemos generar es compleja o real
34      if (cmplx)
35      {
36          printf("***************FFT GPU COMPLEX***************\n");
37          printf("Size: %d\n  NX: %d\n  BATCH: %d\n\n", NX * BATCH, NX, BATCH
    );
38
39          // Creamos el plan para nuestra FFT
40          createPlan(&plan, NX, BATCH, 'C');
41
42          // Generamos los datos de entrada para la FFT
43          float **datos_ = (float**) malloc(sizeof(float*) * NX * BATCH);
44          datos_ = genMatrixComplex(NX, BATCH, SEED);
45
46          // Pasamos los datos a una variable cufftComplex
47          cufftComplex **datos = (cufftComplex**) malloc(sizeof(cufftComplex
    *) * NX * BATCH);
48          for (int i = 0; i < BATCH; i++)
49          {
50              datos[i] = (cufftComplex*) malloc(sizeof(cufftComplex) * NX *
    2);
51              for (int j = 0; j < NX * 2; j += 2)
52              {
53                  datos[i][j / 2].x = datos_[i][j];
54                  datos[i][j / 2].y = datos_[i][j + 1];
55              }
56          }
57
58          // Vectores para almacenar los datos de entrada y salida en forma
    de vector unico
59          cufftComplex *idata = (cufftComplex*) malloc(NX * BATCH * sizeof(
    cufftComplex));
60          cufftComplex *odata_ = (cufftComplex*) malloc(NX * BATCH * sizeof(
    cufftComplex));
61
62          // Variables del device para los datos de entrada y salida
63          cufftComplex *d_idata;
64          cufftComplex *d_data;
65
66          // Reservamos memoria en el device
67          cudaMalloc((void**)&d_data, sizeof(cufftComplex) * NX * BATCH);
68          cudaMalloc((void**)&d_idata, sizeof(cufftComplex) * NX * BATCH);
69
70          // Variable para almacenar el resultado de la FFT
71          cufftComplex **odatos = (cufftComplex**) malloc(sizeof(cufftComplex
    ) * NX * BATCH);
72
73          // Pasamos los datos de una matriz a un vector
74          for (int i = 0; i < BATCH; i++)
75          {
76              for (int j = 0; j < NX; j++)
77              {
78                  idata[j + i * NX] = datos[i][j];
79              }
80              printf("\n");
81          }
82          printf("\n");
83
```

```
84          // Pasamos los datos de entrada del host al device
85          cudaMemcpy(d_idata, idata, sizeof(cufftComplex) * NX * BATCH,
    cudaMemcpyHostToDevice);
86
87          // Comenzamos a grabar el tiempo de ejecucion
88          start = clock();
89
90          // Lanzamos la FFT y obtenemos la medida final del tiempo de
    ejecucion
91          end = vectorFFT_C2C(odatos, plan, NX, BATCH, d_idata, d_data,
    odata_);
92
93          // Liberamos memoria de la GPU
94          cudaFree(d_idata);
95          cudaFree(d_data);
96
97          // Calculamos el tiempo total
98          clock_t total = end - start;
99          printf("Total GPU: %d\n\n", total);
100     }
101     else
102     {
103         printf("***************FFT GPU REAL***************\n");
104         printf("Size: %d\n  NX: %d\n  BATCH: %d\n\n", NX * BATCH, NX, BATCH
    );
105
106         // Creamos el plan para nuestra FFT
107         createPlan(&plan, NX, BATCH, 'R');
108
109         // Generamos los datos de entrada para la FFT
110         float **datos = (float**) malloc(sizeof(float*) * NX * BATCH);
111         datos = genMatrixReal(NX, BATCH, SEED);
112
113         // Vectores para almacenar los datos de entrada y salida en forma
    de vector unico
114         float *idata = (float*) malloc(NX * BATCH * sizeof(float));
115         cufftComplex *odata_ = (cufftComplex*) malloc(NX * BATCH * sizeof(
    cufftComplex));
116
117         // Variables del device para los datos de entrada y salida
118         float *d_idata;
119         cufftComplex *d_data;
120
121
122         // Reservamos memoria en el device
123         cudaMalloc((void**)&d_data, sizeof(cufftComplex) * (NX / 2 + 1) *
    BATCH);
124     cudaMalloc((void**)&d_idata, sizeof(float) * NX * BATCH);
125
126         // Variable para almacenar el resultado de la FFT
127         cufftComplex **odatos = (cufftComplex**) malloc(sizeof(cufftComplex
    ) * (NX / 2 + 1) * BATCH);
128
129         // Pasamos los datos de una matriz a un vector
130         for (int i = 0; i < BATCH; i++)
131         {
132             for (int j = 0; j < NX; j++)
133             {
134                 idata[j + i * NX] = datos[i][j];
135             }
136         }
137
138         // Pasamos los datos de entrada del host al device
```

```
139         cudaMemcpy(d_idata, idata, sizeof(float) * NX * BATCH,
    cudaMemcpyHostToDevice);
140
141         // Comenzamos a grabar el tiempo de ejecucion
142         start = clock();
143
144         // Lanzamos la FFT y almacenamos la medida final del tiempo de
    ejecucion
145         end = vectorFFT_R2C(odatos, plan, NX, BATCH, d_idata, d_data,
    odata_);
146
147         // Liberamos memoria de la GPU
148         cudaFree(d_idata);
149         cudaFree(d_data);
150
151         // Calculamos el tiempo total
152         clock_t total = end - start;
153         printf("Total GPU: %d\n\n", total);
154     }
155     return 0;
156 }
```

La salida es la misma que en el código de pruebas de Intel IPP. Únicamente se diferencia en que pone GPU en vez de CPU y no se imprime la parte de *Using FFT* o *Using DFT*.

```
***************FFT GPU COMPLEX***************
Size: 20480
  NX: 320
  BATCH: 64

Total GPU: 226
```

7.2.3. Ejecución de las pruebas, resultados y conclusiones

De momento tenemos claro que el rendimiento de las FFT de ambas librerías depende principalmente de si el tamaño del vector de potencia de dos o no. Sin embargo, no sabemos como cambia en función de la cantidad de puntos y de la cantidad de transformadas que tengan que hacer.

Para el caso de Intel IPP se puede intuir que el crecimiento del tiempo de ejecución crece linealmente en función de la cantidad de transformadas, pero no está claro si también es un crecimiento lineal con el aumento del número de puntos.

Por otro lado, no está tan claro con cuFFT. Sabemos que la GPU admite una cierta cantidad de datos sin apenas modificar el tiempo de ejecución, debido a la paralelización. Además, se introducen todos los vectores de una vez, en vez de uno en uno como en Intel IPP, y no sabemos qué tipo de gestión hace la librería en función de la cantidad de estos.

Por último, tampoco tenemos una comparación del rendimiento de ambas librerías para el caso que nos interesa en este proyecto. Que son una gran cantidad de transformadas con una cantidad de puntos relativamente pequeña.

La principal razón de estas pruebas fue aclarar las dudas planteadas. También se pretendió aprender el funcionamiento de ambas librerías de una forma más profunda que simplemente

leyendo la documentación.

Las prueban se dividieron en tres partes, con la intención de cubrir todos los casos necesarios. En cada una se han hecho pruebas tanto con entradas reales como complejas. Se decidió hacer diez ejecuciones para tipo de entrada en cada prueba. Es decir, por cada prueba se ha lanzado el programa diez veces para la entrada real y otras diez para la compleja. Posteriormente se ha realizado un filtrado de datos para evitar medidas anómalas. Este filtrado consiste en eliminar los datos que se alejen más de tres desviaciones típicas. La razón de tomar un rango tan grande se basa en que el tiempo de ejecución en estas pruebas es bastante variable, pudiendo ser correcta una medida que a priori parezca anómala. Finalmente, se pintan las gráficas de la forma más adecuada para la visualización de los datos.

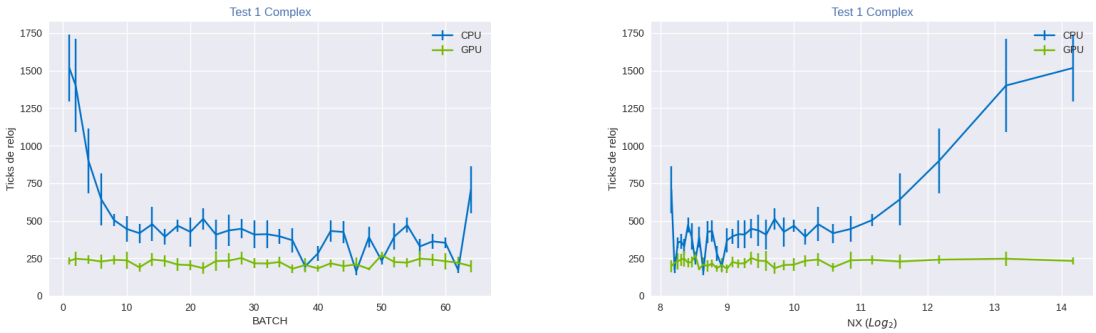
Las pruebas no se han hecho manualmente, sino que se han programado utilizando el lenguaje de scripting BASH. El procesado de los resultados y la generación de las gráficas se han realizado con Python.

- **Primer test**

  Es un test bastante general que pretende ver el comportamiento de las librerías cuando el NX baja a la vez que el BATCH sube. Para ello se ha tomado un vector inicial de 18.420 puntos. Este vector se va dividiendo en varias partes hasta llegara 64 partes. Esta división se realiza de dos en dos, es decir, primero tenemos un único vector con todos los puntos, luego dos vectores con la mitad cada uno, después cuatro vectores con un cuarto cada uno, etcétera.

  En las figuras 7.7a y 7.7b se pueden ver los resultados para entrada compleja y las figuras 7.8a y 7.8b para entrada real.

  Hay que tener en cuenta que tanto la variable BATCH como NX son distintas en cada punto. Para las gráficas con el BATCH en el eje de abscisas, NX será mayor a menor valores del BATCH y viceversa. Para las gráficas con NX en el eje de abscisas ocurre algo similar. Para NX más grandes tendremos BATCH más bajos y viceversa.

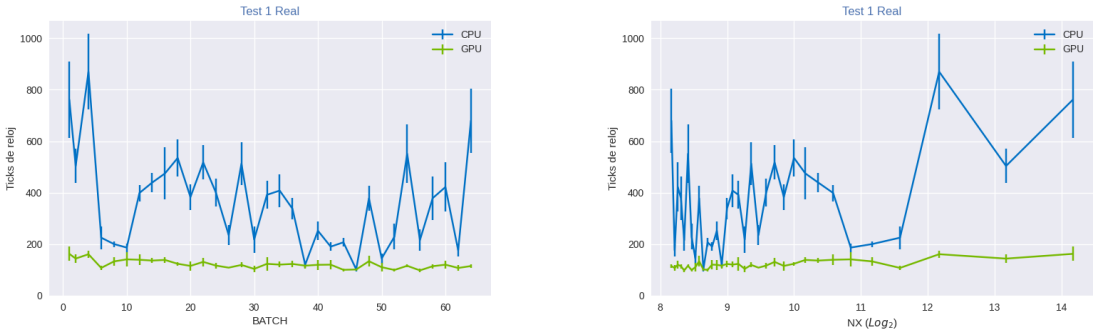


(a) BATCH en el eje de las abscisas. (b) NX en el eje de las abscisas.

Figura 7.7: Resultados del test 1 para entrada compleja.

Podemos observar un mejor rendimiento de la GPU en cualquiera de los casos. Aunque en las partes en las que la cantidad de BATCH aumenta y NX disminuye, el rendimiento de Intel IPP mejora bastante, llegando a igualar a cuFFT.

Aún no podemos sacar conclusiones muy contundentes pero podemos empezar a entender



(a) BATCH en el eje de las abscisas. (b) NX en el eje de las abscisas.

Figura 7.8: Resultados del test 1 para entrada real.

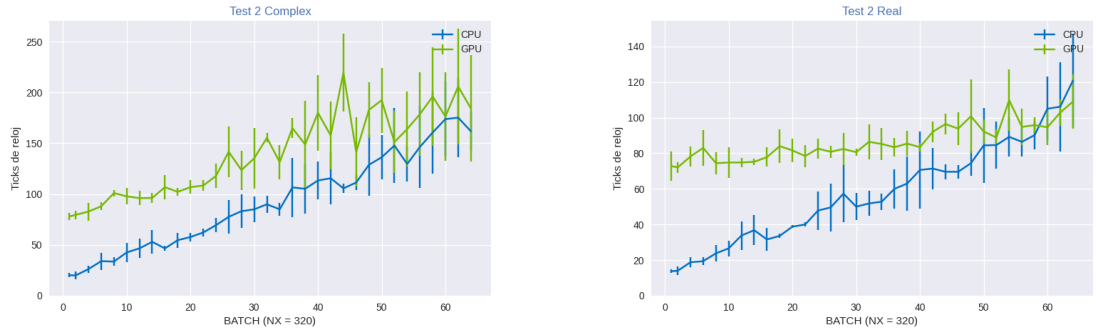
el comportamiento de ambas librerías. Con cuFFT podemos ver que se mantiene muy estable en todos los puntos, de lo que podemos deducir que el rendimiento depende bastante del número de puntos total, es decir, el resultado de hacer $NX \cdot BATCH$. Sin embargo, Intel IPP sufre muchas más variaciones, siendo más irregulares con entrada real. Aún así se puede ver que está muy afectada por el NX que por BATCH. Es más, para BATCH más bajos mejores rendimiento de Intel IPP tenemos, debido a que NX va disminuyendo.

- **Segundo test**

  En este test se pretende ver el comportamiento cuando NX es fijo y el BATCH se va variando desde 1 hasta 64, cogiendo los números pares y el uno. Se han escogido dos valores para dicho NX fijo: 320 y 512. El primero número es la cantidad de puntos utilizada en el experimento que se ha tomado como ejemplo, el *vo1021*, y el segundo número es debido a que es la siguiente potencia de dos por encima de 320. Del experimento se ha tomado una muestra, denominada *scan*, que corresponde a una de las múltiples observaciones que se llevan a cabo en la realización de un experimento.

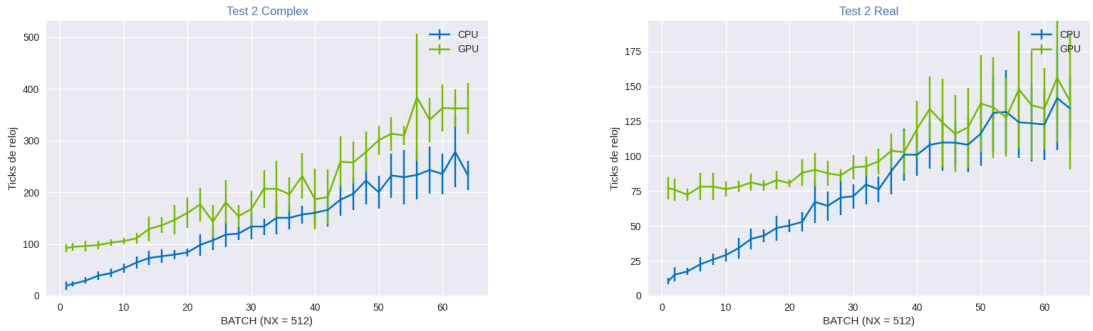
  Para cada NX se han hecho dos pruebas diferentes, una para entrada real y otra para entrada compleja, siguiendo el mismo concepto de los test anteriores.

  Los resultados para un NX de 320 se pueden ver en las figuras 7.9a para entrada compleja y 7.9b para entrada real, y para un NX de 512 en las figuras 7.10a para entrada compleja y 7.10b para entrada real.



(a) Entrada compleja. (b) Entrada real.

Figura 7.9: Resultados del test 2 con NX fijo en 320.



(a) Entrada compleja.

(b) Entrada real.

Figura 7.10: Resultados del test 2 con NX fijo en 512.

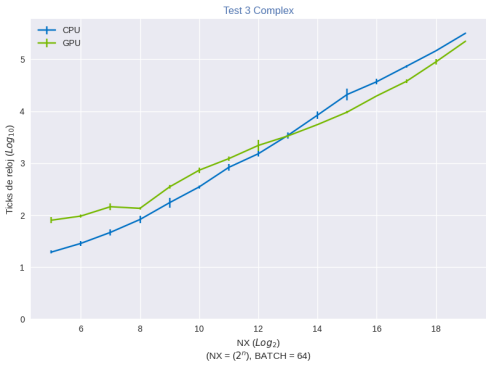
En las gráficas se puede apreciar como esta vez la GPU está por encima en todos los casos, como era de esperar según las deducciones sacadas con los resultados de las anteriores pruebas. Con cantidades de puntos pequeñas, Intel IPP es capaz de desenvolverse mejor que cuFFT.

- **Tercer test**

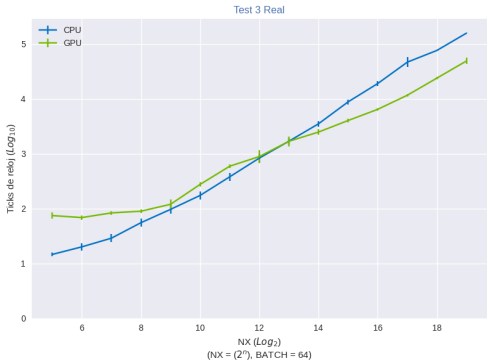
Por último, en este test se pretende ver el rendimiento cuando se utiliza un BATCH fijo y es el NX el que varía. Se ha escogido como 64 para el BATCH tomando como referencia los datos de la correlación del experimento *vo1021*. A la hora de configurar una correlación, el BATCH representa el número de canales de frecuencia, que suele estar en torno al número elegido.

Respecto al NX, primero se han hecho pruebas con 2^n puntos con $5 \leq n \leq 20$. Con esto se pretende que ambas librerías estén en su máximo rendimiento. Después se ha cambiado la forma de generar la cantidad de puntos para que ya no sean múltiplos de dos. Esta nueva forma es 2.007^n con $5 \leq n \leq 20$. La decisión de coger como base un número muy cercano a dos es que NX sea muy parecido a una potencia de dos para poder comparar los resultados con la otra prueba. De esta forma también podemos ver la diferencia dentro de una misma librería entre hacer la transformada aprovechando al máximo su algoritmo y sin aprovecharlo, es decir, entre la FFT y la DFT.

Los resultados para NX potencia de dos se pueden ver en las figuras 7.11a para entrada compleja y 7.11b para entrada real, y para NX que no es potencia de dos en las figuras 7.12a para entrada compleja y 7.12b para entrada real.

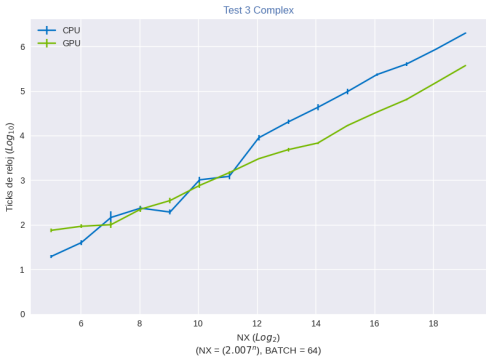


(a) Entrada compleja.

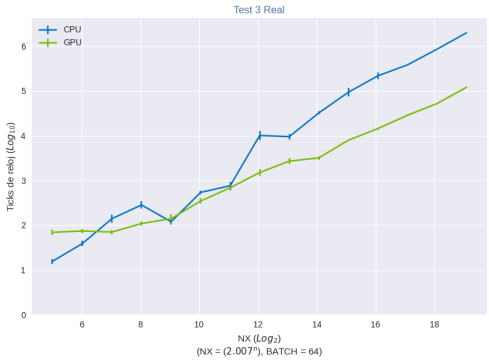


(b) Entrada real.

Figura 7.11: Resultados del test 3 con $NX = 2^n$.



(a) Entrada compleja.



(b) Entrada real.

Figura 7.12: Resultados del test 3 con $NX = 2.007^n$.

Viendo los resultados de esta prueba se puede ver, tal y como dedujimos en el primer test, que la GPU mejora su rendimiento frente a la CPU cuando la cantidad de puntos por

vector es alta. Este punto está en aproximadamente 2^{13} puntos en los casos en los que se ha utilizado la FFT y en 2^{11} puntos en los casos en los que se ha utilizado la DFT. Como podemos ver, existe una notable diferencia entre utilizar la FFT o la DFT. Mientras que cuFFT no presenta cambios muy notables respecto a utilizar NX que sean potencia de dos o no, Intel IPP sí que ve aumentado su rendimiento de forma más notable.

Como conclusión final se puede decir que cuFFT soporta mejor el aumento de la cantidad de puntos por vector mientras que Intel IPP alcanza mejores rendimiento con vectores más pequeños.

Sin embargo, esta conclusión se ve afectada por la manera en la que hemos medido el tiempo en cada prueba. En el código de la GPU se ha incluido dentro de este tiempo el paso de los datos tanto del host al device como del device al host. Además del tiempo que se tarda en convertir la matriz de entrada a un vector. Esta decisión viene porque si se implementa cuFFT en DiFX estas operaciones son necesarias. Por otro lado, no se ha incluido el tiempo de volver a pasar el vector salida a una matriz debido a que se puede incluir dentro de la propia estructura de DiFX sin tener que hacer bucles adicionales.

Es por ello que esta comparativa está condicionada a la estructura de DiFX y no refleja unas pruebas reales de rendimiento de Intel IPP y cuFFT. Pero son suficientes para el interés de este proyecto.

Capítulo 8

Diseño de las modificaciones

En este punto del proyecto ya se ha obtenido el conocimiento necesario para acceder al código y empezar a estudiar, con el objetivo de diseñar las modificaciones que se pretenden implementar. Estas modificaciones consisten en sustituir todo lo relativo a Intel IPP en la realización de la Transformada de Fourier por las funciones de cuFFT. Las pruebas relativas a la FFT realizadas en el capítulo 7 sirven como guía para lo que se pretende hacer.

Los ficheros que se encargan de procesar los datos de la correlación son *core.cpp* y *mode.cpp*. Con un análisis más en detalle se ve que desde las operaciones que se buscan están en mode, y que *core.cpp* únicamente se encarga de llamar a la función que las contiene. Por lo que descartaremos a *core.cpp* y nos centraremos en *mode.cpp*.

Dentro de *mode.cpp* se encuentran varias funciones, aparte del constructor, necesarias para el correcto funcionamiento de las operaciones realizadas por el Core. Estas operaciones son, concretamente, la Transformada de Fourier, la rotación de franja y la corrección del retardo fraccionario de las muestras, y que ya han sido explicadas en el capítulo 6. Estas operaciones se llevan a cabo dentro de una sola función de *mode.cpp*. que es *process()*. El resto de funciones no tienen ninguna participación en la operación de la Transformada de Fourier, por lo que se obviarán.

En este aparado se mostrará la estructura de dicha función, así como los cambios en esta para que DiFX sea capaz de ejecutar la Transformada de Fourier utilizando cuFFT en vez de Intel IPP.

8.1. Estructura de la función *process*

Una vez identificada esta función, hay que estudiar en profundidad su estructura para ir identificando cada una de las operaciones anteriormente mencionadas. No se va a exponer el código debido a que es muy extenso y hay partes donde el interés está en su funcionalidad y no en las líneas de código. En su caso, se va a utilizar el diagrama 8.1, que agrupa las partes principales y que refleja el flujo de ejecución.

Antes de entrar en detalle hay que tener conocimiento sobre algunas variables relevantes. Podemos ver que de momento hay dos: *numrecordedfreqs* y *numrecordedbands*. La primera corresponde a la cantidad de frecuencias grabadas en un experimento mientras que la segunda a la cantidad de canales utilizados. Cada canal corresponde a una frecuencia y a una polarización.

En algunos experimentos se graban ambas polarizaciones por lo que no tiene porqué coincidir el número de frecuencias con el número de canales.

Al inicio de la función se inicializan y se reserva memoria para las variables que se utilizarán más adelante. Después se obtienen los datos, proporcionados por el nodo DataStream, y se almacenan en la variable *unpackedarrays* o *complexunpacked* si son números complejos. Estas variables son dobles punteros, es decir que son matrices, donde cada fila será un vector de entrada para la FFT.

Con todas las variables inicializadas y los datos preparados, se pasa al primer bucle, limitado por la variable *numrecordedfreqs* y que se encarga de recorrer todas las frecuencias del experimento. Para cada iteración, primero se prepara la exponencial compleja que se utilizará para la rotación de franja. Esta exponencial dependerá de si se ha escogido la opción pre-F o post-F. Después se entra en el segundo bucle, limitado por *numrecordedbands* y que recorre todos los canales. En cada iteración se comprueba si el canal correspondiente a la variable j corresponde con la frecuencia que se está tratando y que corresponde a la variable i. Esta comprobación se realiza con una llamada a una función alojada en otro fichero destinado a la configuración de la correlación. Sin embargo, en el diagrama se ha representado como $i = j$ por simplificación.

Una vez tenemos el canal correspondiente para la frecuencia tratada, el siguiente paso es ver si se ha configurado la correlación como pre-F o post-F. La opción pre-F quiere decir que la corrección de franja se realiza antes de la FFT y la opción post-F significa que la corrección de franja se realiza después. Dependiendo de la opción elegida, la exponencial compleja utilizada en la corrección de franja se realizará en el dominio del tiempo o en el dominio de la frecuencia, como se explicó en el capítulo 6 en el apartado *"Rotación de franja"*.

Después de aplicar la FFT y la rotación de franja, con el orden correspondiente, se realiza la corrección del error fraccionario. Este error es el retardo que faltaba por corregir al realizar el alineamiento que corrige el *integer delay*.

La salida de la FFT se almacena en los vectores *fftptr* para el caso de pre-F y *fftd* para el caso post-F. Almacenan una única salida, por lo que en cada iteración del bucle donde se realiza la FFT hay que guardar este resultado en otra variable, llamada *fftoutputs*. Esta es un triple puntero que almacena los resultados de todas las operaciones que se realizan en la función, y que serán tomados por el siguiente proceso del correlador.

Es importante conocer bien el funcionamiento de las variables mencionadas, debido a que son las entradas y salidas de la FFT. A continuación se muestra la declaración de estas variables, indicando su tipo, y sus análogas utilizando los tipos de cuFFT. Se ha obviado a *fftoutputs* debido a que es la variable que utilizará el resto de las funciones del correlador y no es posible hacer su correspondiente versión para cuFFT. Los tipos de las variables están renombradas según la tabla 8.1.

```
// Entrada de datos usando Intel IPP
f32     ** unpackedarrays;
cf32    ** complexunpacked;

// Entrada de datos usando cuFFT
f32_c    * cuda_unpackedarrays
cf32_c  ** cuda_complexunpacked;
cf32_c   * cuda_complexunpacked_vector;

// Salida de datos usando Intel IPP
cf32    * fftptr;
```

```
12 cf32    * fftd;
13
14 // Salida de datos usando cuFFT
15 cf32_c  * cuda_fftptr;
16 cf32_c  * cuda_fftd;
```

Nombre utilizado	C++	Nombre original
f32_c	float	cufftFloat
f32		Ipp32f
cf32_c	complex	cufftComplex
cf32		Ipp32fc
cf64_c	doubleComplex	cufftDoubleComplex
cf64		Ipp64fc

Tabla 8.1: Nombre de los tipos de variable utilizado para sustituir los nombres originales utilizados por Intel IPP y cuFFT.

Por último, antes de pasar a ver las modificaciones para post-F y para pre-F, es necesario explicar como se ha realizado la gestión del plan necesario para la FFT.

8.2. Gestión del planificador de una FFT

Este planificador es el que le indica a la función de la FFT el tamaño de los vectores de entrada, la cantidad de vectores y el tipo de transformada. Crear este plan es bastante costoso en términos de tiempo computacional, lo que representa un problema si lo que intentamos es acelerar el correlador.

La ventaja es que un plan puede ser utilizado para realizar todas las transformadas que se deseen siempre que se mantenga la configuración. Esto se puede aprovechar, evitando ese gasto adicional de tiempo.

La mejor manera de hacer que varios procesos accedan a una única variable es utilizando un singleton. Esta técnica consiste en crear un objeto estático, es decir que sus métodos pueden ser llamados sin necesidad de tener un objeto de esa misma clase, que contenga una instancia del objeto que se desea compartir. Cada proceso que quiere utilizar este recurso deberá pedirlo a la clase, y esta devolverá dicho recurso. De esta forma solo tenemos un único plan que puede ser utilizado por todos los procesos que lo requieran.

El principal problema que se presenta es la concurrencia que tienen los Cores de DiFX, generando conflictos a la hora de acceder a una única variable. La solución para este tipo de problemas de acceso de recursos por hilos concurrentes se puede solucionar con bloqueos en las funciones. Cuando un hilo llama a la función para obtener una instancia del plan establecerá un bloqueo que impida a los demás hilos entrar a la función. Cuando consiga el plan quitará el bloqueo, permitiendo que pase el siguiente hilo.

Existe un inconveniente para esta solución y es que se pierde parte de la concurrencia al hacer que los hilos tengan que acceder de forma secuencial a un determinado recurso. Sin embargo, esta solución es bastante más rápida que generar un plan para cada FFT que se quiera realizar.

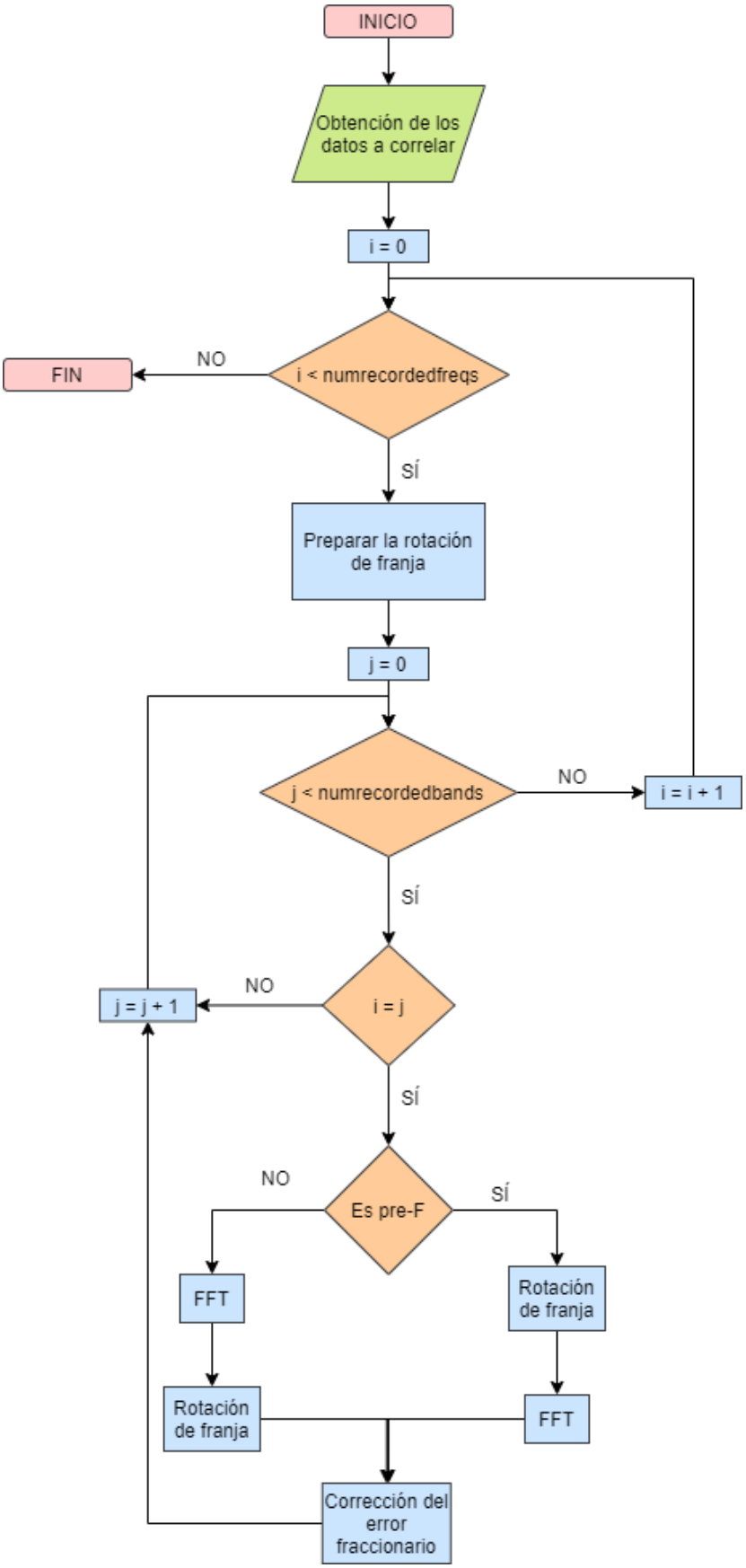


Figura 8.1: Diagrama de flujo de la función *process*.

8.3. Modificaciones para el caso post-F

Este es el caso más sencillo de los dos, debido a que la rotación de franja y la corrección del error fraccionario se realizan en el mismo punto. Si fuésemos fieles al diagrama 8.1 y se siguiese el camino correspondiente a pre-F nos quedaría que la FFT se realiza dentro de los bucles anidados. Sin embargo, la principal característica de cuFFT es que es capaz de hacer varias FFT en una sola llamada.

Si dejásemos la FFT dentro de los bucles tendríamos que hacer una sola FFT por iteración, perdiendo esta característica de cuFFT. La solución sería sacar la operación fuera, concretamente en el punto antes de entrar el primer bucle, tal y como se ve en el diagrama 8.2.

Las variables para almacenar la salida de la FFT ya no se pueden utilizar, debido a que son vectores enfocados a guardar la salida en cada iteración, y ahora se obtienen todos los vectores de una vez, por lo que se utiliza una variable auxiliar que haga como intermediaria entre la FFT y la rotación de franja.

Esta variable es *cuda_fftptr*, que es un puntero, es decir, un vector. Esto puede resultar confuso debido a que en realidad la salida de la FFT es una matriz. Como se ha comentado en capítulos anteriores, las tarjetas gráficas trabajan de forma más eficiente con vectores en vez de con matrices, por lo que las matrices se transforman en vectores, concatenando sus filas o sus columnas, según sea necesario.

Una vez se ha llevado a cabo la FFT ya se puede pasar a los bucles anidados. Dentro de estos, en el punto en el que antes se hacía la FFT ahora se copia el vector correspondiente a la variable *fftptr*, que era la original que guardaba los resultados de las transformadas. Esta variable es un vector capaz de almacenar únicamente el resultado de una sola FFT. En cada iteración se copia el vector correspondiente, en función del canal que se esté tratando.

En los siguientes pasos se pasarán los datos de *ffptr* a *fftoutputs* y se llevará a cabo la rotación de franja y la corrección del error fraccionario de forma normal.

8.4. Modificaciones para el caso pre-F

En este caso la rotación de franja se realiza antes de la FFT. Esto significa que la rotación de franja y la corrección del error fraccionario están separadas por la FFT. Este es un gran problema debido a que, según el diseño de *process*, ambas operaciones necesitan realizarse dentro de dos bucles anidados.

Sin embargo, la FFT no necesita estar dentro de ningún bucle. La solución proporcionada para este problema es dividir el bucle en dos partes. Primero para llevar a cabo la corrección de franja y segundo para la corrección del error fraccionario.

Este caso no se ha podido representar en un solo diagrama por sus dimensiones, por lo que se ha dividido en varias partes. Por un lado tenemos lo que se ha denominado bloques de rotación de franja (8.3) y de corrección del error fraccionario (8.4), y por el otro el diagrama principal (8.5).

Primero se realizan los bucles encargados de preparar la rotación de franja y de llevarla a cabo. Los resultados de esta operación se guardan en la misma variable que almacenaba los datos de los experimentos, *unpackedarrays* o *complexunpacked* dependiendo del tipo de datos.

Después se terminan los dos bucles y se pasa a la FFT. La entrada sigue siendo la matriz (pasada a vector) original, ya que no se ha creado ninguna nueva para la salida de la rotación de franja. La salida, por otro lado, se almacena en una variable diferente. Este es el mismo caso que en post-F, originalmente los resultados se iban almacenando en un vector según se realizaban las transformadas, pero ahora hacen todas de una vez. La solución también ha sido similar a la de post-F. Se ha utilizado una nueva variable llamada *cuda_fftd*, que es un vector que almacena una matriz.

El siguiente bloque, el de la corrección del error fraccionario, también se compone de dos bucles anidados. Dentro de estos, aparte de realizarse la operación mencionada, también se copian los datos de la variable auxiliar *cuda_fftd* a la original *fftd*.

En los siguientes pasos se irán pasando los datos de *fftd* a *fftoutputs*.

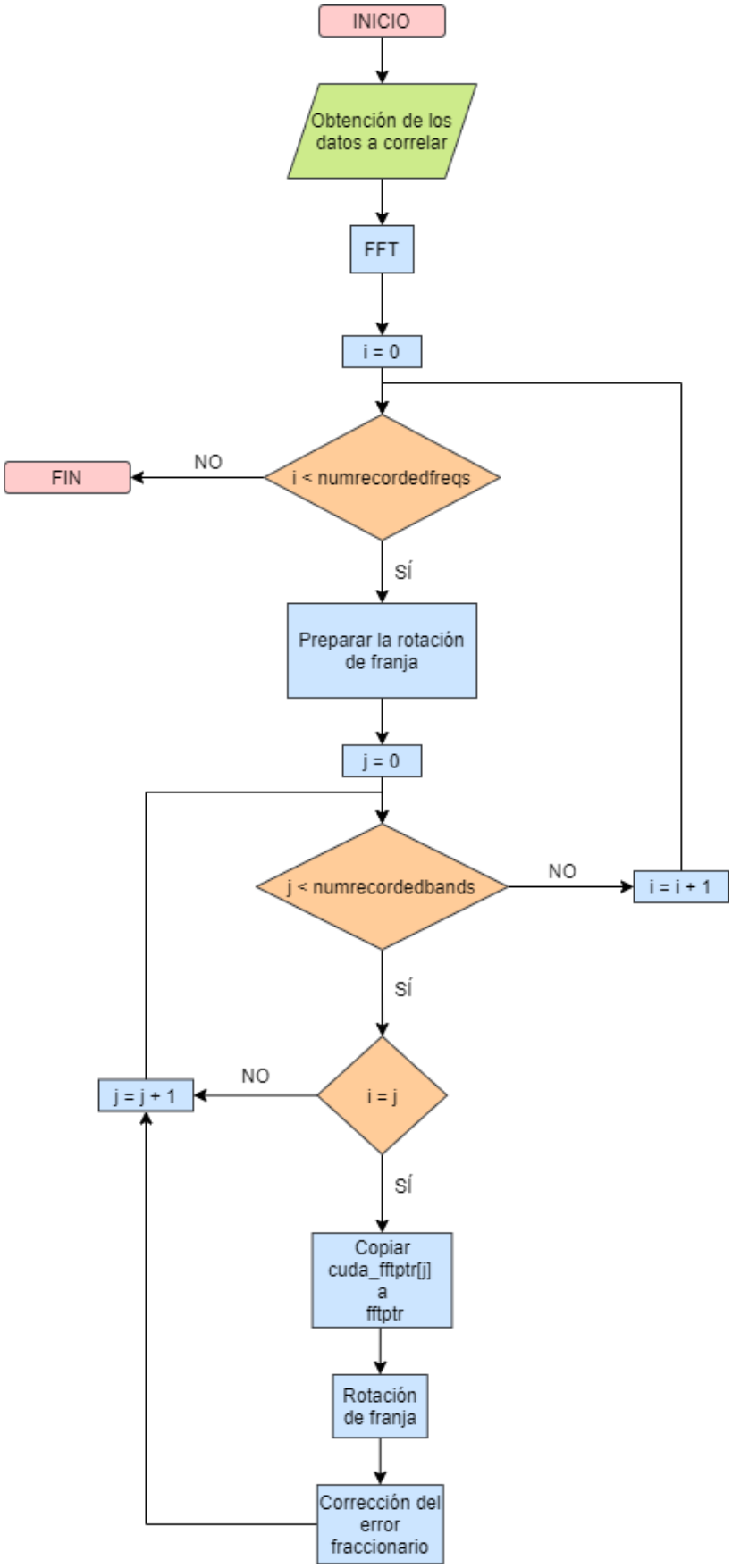


Figura 8.2: Diagrama de flujo de la función *process* con los cambios de cuFFT para el caso post-F.

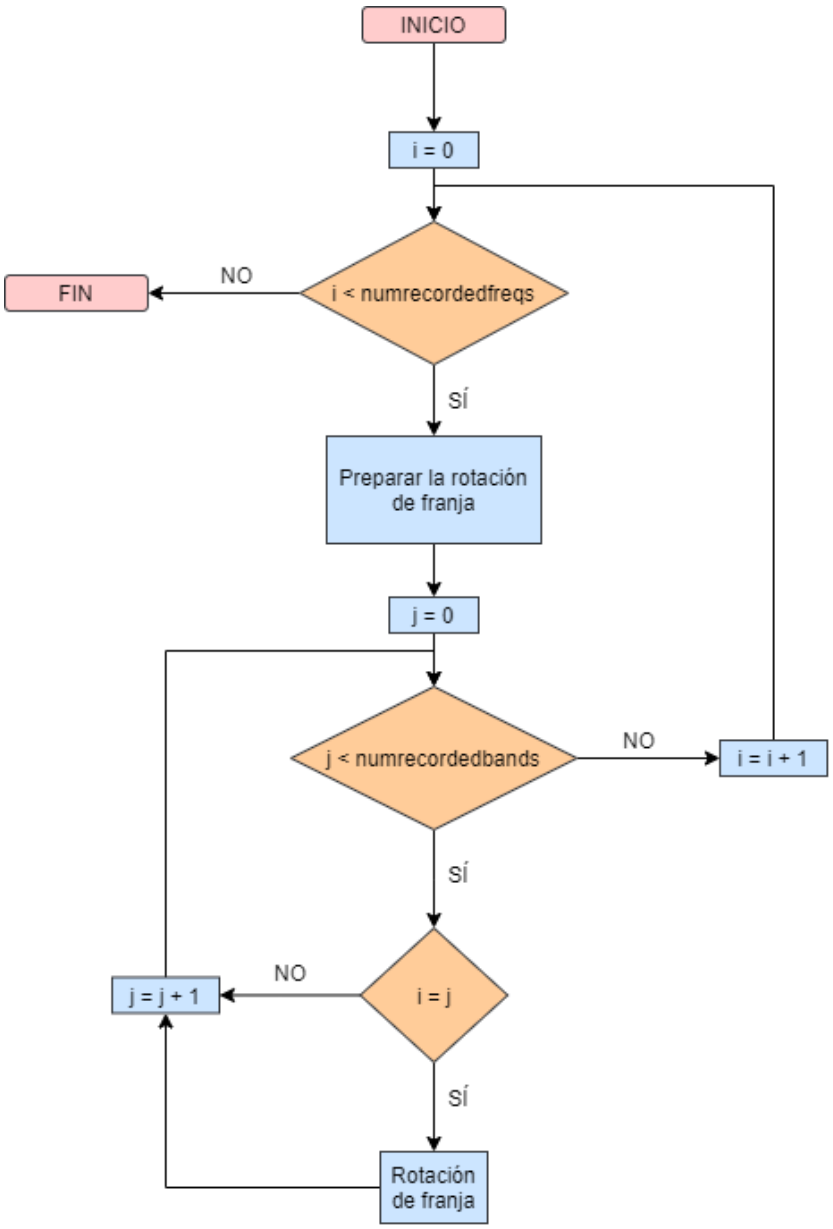


Figura 8.3: Bloque donde se lleva a cabo la rotación de franja.

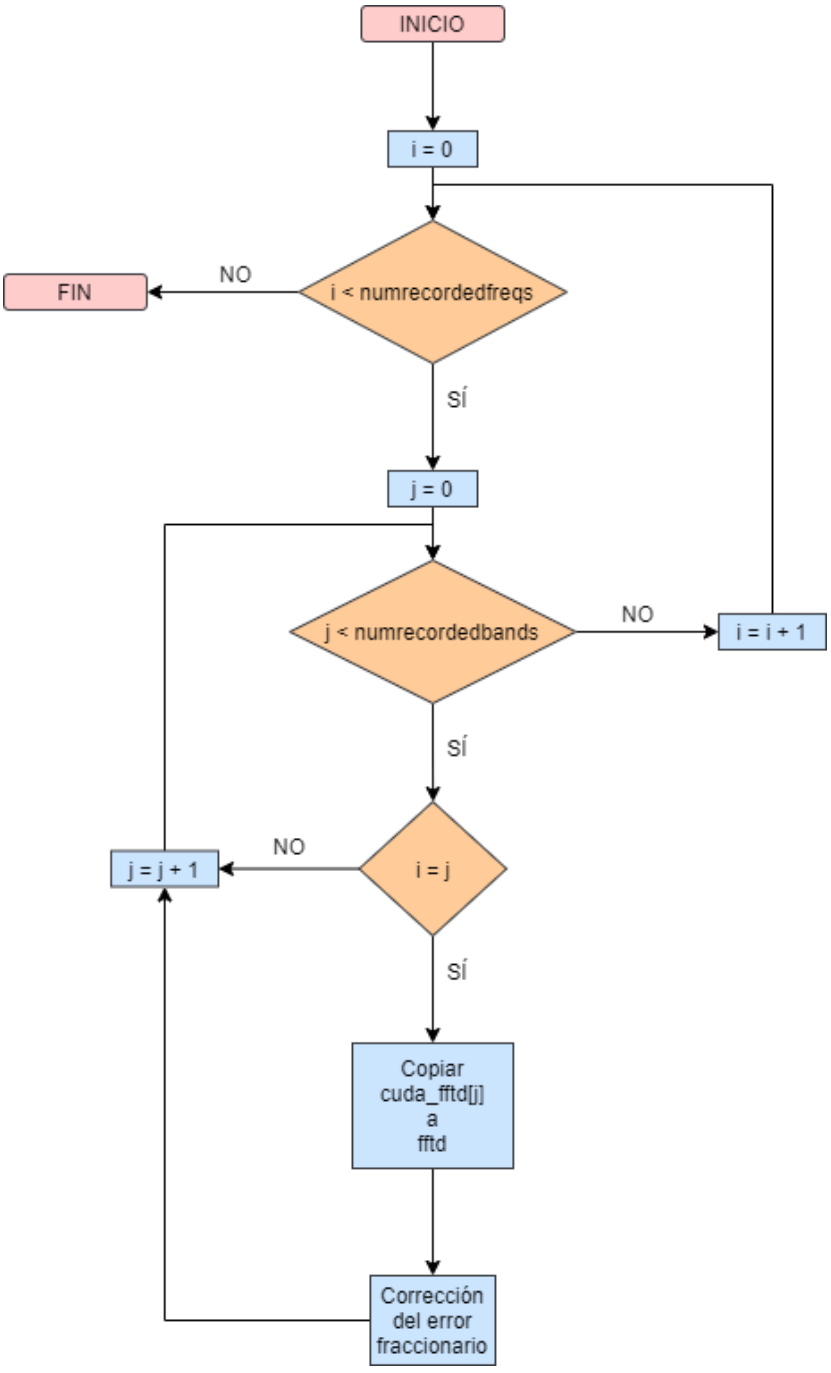


Figura 8.4: Bloque donde se lleva a cabo la corrección del error fraccionario.

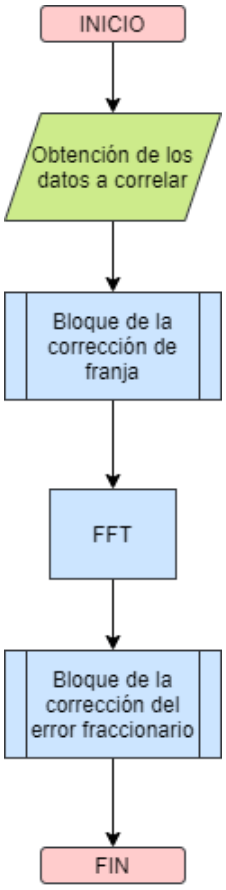


Figura 8.5: Diagrama de flujo para el caso pre-F. Los bloques de la rotación de franja (8.3) y de la corrección del error fraccionario (8.4) se exponen a parte.

Capítulo 9

Implementación de las modificaciones

Ya se ha creado el diseño de las modificaciones que se quieren implementar. Además, las pruebas realizadas con cuFFT y el código programado en estas se cree suficiente como para poder implementar estas modificaciones. Sin embargo, no ha sido posible realizar la implementación en su plenitud.

El primer caso que se quiso programar es el post-F, debido a que es el más sencillo. Primero se hizo una prueba dentro del propio código de DiFX que consistía en realizar una FFT sencilla, de cuatro puntos, e imprimir el resultado. El objetivo era comprobar que funcionaban toda las funciones de cuFFT y todos los cambios realizados en el archivo que define la arquitectura del fichero: *architecture.h*. En este fichero es donde se renombran los tipos de las variables y los nombres de las funciones. También existen funciones con código ejecutable, en vez de macros, como puede ser la ejecución de una FFT. Estas funciones son exactamente las mismas que se utilizaron en las pruebas del capítulo 7, en el apartado de cuFFT.

Tras comprobar que todo funcionaba correctamente se pasó a la modificación del fichero. Se implementaron los cambios tal y como se han expuesto en el capítulo 8. El código programado sigue la misma estructura que el de las pruebas de cuFFT, con los únicos cambios del nombre de las variables.

Finalmente, y tras resolver varios fallos directos del nuevo código implementado, se intentó comprobar el correcto funcionamiento de este. Sin embargo, aparecieron diversos problemas que en primera instancia no tenían una relación directa con el nuevo código y que no se consiguieron resolver.

La razón de esto viene de la complejidad del código de DiFX. En este proyecto se ha analizado en profundidad un fichero muy concreto y un código muy concreto, y aun así no se llegó a un entendimiento completo de este. Esta dificultad y la falta de tiempo para el desarrollo de los objetivos iniciales del proyecto resultó en la incapacidad de resolver los problemas comentados.

Por otro lado, el caso de pre-F posee una complejidad aún mayor. También se llevó a cabo la implementación pero, al igual que para post-F, no se le pudo dedicar el suficiente tiempo como para hacer funcionar estas modificaciones.

Capítulo 10

Conclusiones y trabajo futuro

El objetivo inicial del proyecto era acelerar el correlador DiFX utilizando técnicas de programación en GPUs y aunque no se haya alcanzado, las pruebas y el estudio del código han sido suficientes como para poder sacar unas conclusiones relevantes.

Al trabajar con GPUs en lugar de con CPUs para las principales operaciones del Core, que es lo que se quería conseguir originalmente, se buscaban los siguientes objetivos:

- Mejorar el tiempo necesario para correlar los experimentos.
- Intentar progresar hacia las correlaciones de las observaciones en tiempo real.
- Hacer un estudio del camino a seguir para adaptar DiFX al uso de tarjetas gráficas.

Podríamos considerar que este proyecto ha cubierto dos de estos tres objetivos. Con el análisis del código y de las librerías, que mejor se adaptan a la resolución de las operaciones implicadas en una correlación, se ha conseguido proporcionar una visión más detallada de los inconvenientes que surgen a la hora de llevar los correladores implementados en un clúster únicamente con CPUs a otros que puedan implementar GPUs.

El segundo objetivo cubierto es el progreso hacia las correlaciones en tiempo real, que viene dado al cumplir el objetivo que se acaba de explicar.

Los inconvenientes que se han observado son, en su mayoría, la diferencia que existe entre la forma de operar de una CPU y la de una GPU, como pueden ser:

- Las CPUs son mejores cuando la cantidad de datos es relativamente pequeña y la complejidad de las operaciones es grande, y las tarjetas gráficas alcanzan su mayor potencial al trabajar con cantidades de datos grandes y operaciones sencillas.
- Los procesadores pueden hacer operaciones directamente sobre matrices, mientras que las GPUs necesitan que los datos estén en vectores para que se pueda aprovechar todo su potencial. Si se combina una CPU con una GPU para llevar a cabo operaciones sobre matrices habrá que estar convirtiendo estas en vectores y viceversa continuamente, perdiendo mucho tiempo.

Estos inconvenientes toman importancia debido a que el correlador DiFX está diseñado y estructurado enteramente para trabajar en un clúster con CPUs como única unidad de procesamiento.

Finalmente se plantean dos soluciones que permiten evitar estos inconvenientes, pero que aumentan la complejidad del proyecto.

En primer lugar, en vez de modificar únicamente la parte relativa a la Transformada de Fourier, se podría cambiar la función en su totalidad para que se ejecute enteramente en la GPU. De esta forma se evitan los tiempo del paso de datos entre el host y el device, además de poder mantener los datos en un formato determinado (en un solo vector) durante toda la ejecución. Esta solución sería directa continuación de este proyecto.

En segundo lugar, se puede ir más lejos y, en vez de cambiar una única función, se podría modificar la forma de trabajar de nodo Core. Lo que se propone en este caso es crear un sistema híbrido, donde los procesadores se dediquen a las tareas de los DataStream, enviando datos de los ficheros de almacenamiento a los Cores, y las tarjetas gráficas de las tareas de los Cores, realizando las operaciones matemáticas que se requieren en la correlación. Como se puede intuir, la complejidad de este proyecto aumenta considerablemente, ya que habría que reestructurar el correlador en su totalidad.

Apéndice A

Teorema de Wiener-Khinchin

El teorema de Wiener-Khinchin relaciona la potencia espectral de una señal con la función de correlación. Esta relación permite realizar la correlación de dos señales sin tener que aplicar la función de correlación, siendo más sencillo de realizar en términos computacionales.

La función de correlación, usada en radiointerferometría, se puede escribir como

$$r(\tau) = \lim_{T\to\infty} \frac{1}{2T} \int_{-T}^{T} V_1^*(t) V_2(t-\tau) dt \tag{A.1}$$

donde el asterisco indica la conjugada compleja. Las señales son V_1 y V_2, la primera es la referencia y la segunda la que se desplaza en el tiempo para que coincidan. Este desplazamiento corresponde a τ, que representa el retardo. El resultado de esta ecuación es la función que describe el retardo entre las dos señales, alcanzando su máximo en 0 si el retardo introducido es el correcto.

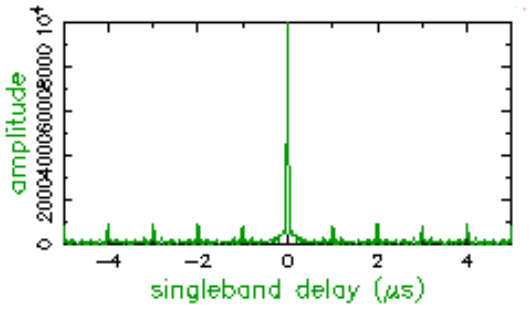


Figura A.1: Resultado de la autocorrelación de una señal

Dadas dos señales v_1 y v_2, la densidad espectral de potencia conjunta S_{v1v2} se define como la multiplicación de la transformada de Fourier de ambas.

$$S_{v1v2} = V_1(\omega) \cdot V_2(\omega) \tag{A.2}$$

Si a S_{v1v2} le aplicamos la Transformada de Fourier inversa el resultado será la función de retardos $r(\tau)$ que obteníamos con la función de correlación. La relación entre la potencia

espectral, la función de correlación y la transformada de Fourier de una señal $x(t)$ se expone en la figura A.2. Esta relación se conoce como el Teorema de Wiener-Khinchin.

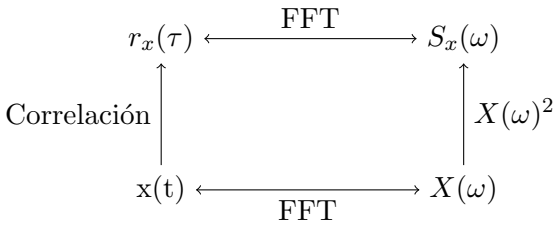


Figura A.2: Relación entre la función de correlación $r_x(\tau)$, la potencia espectral $S_x(\omega)$, la señal original $x(t)$ y su transformada de Fourier $X(\omega)$.

Donde $x(t)$ es la señal original, $X(\omega)$ es la transformada de la señal original, $S_x(\omega)$ son las visibilidades, que representan lo observado y $r_x(r)$ es la función de correlación.

Apéndice B

Transformada de Fourier y Transformada Rápida de Fourier

Este apéndice tiene la intención de proporcionar una breve explicación de la Transformada de Fourier y la Transformada Rápida de Fourier(FFT). No se va a realizar una amplia explicación de las matemáticas que hay detrás de la transformada, para una mayor profundización ver Champeney(1973)[9].

B.1. Transformada de Fourier

La Transformada de Fourier es una transformación matemática empleada para transformar señales entre el dominio del tiempo y el dominio de la frecuencia y se puede aplicar a una gran variedad de problemas dentro de la física y la ingeniería. Aquí se va a exponer su uso para transformar una señal de radio procedente del cielo en el dominio del tiempo a una señal en el dominio de la frecuencia, haciendo que sea posible interpretar esta información.

Antes de explicar la Transformada de Fourier hay que entender la Serie de Fourier. Esta serie nos permite descomponer una señal periódica en una suma infinita de funciones sinusoidales mucho más simples (como combinación de senos y cosenos con frecuencias enteras). Siendo $f(t)$ la función que describe nuestra señal con variable real t, la serie de Fourier es:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{N}(a_n cos(\omega_n t) + b_n sin(\omega_n t)) \tag{B.1}$$

donde a_0, a_n y b_n son los coeficientes de Fourier y $\omega_n = n\omega$ y $\omega = 2\pi\nu$.

Para entender mejor la serie de Fourier, se va a mostrar un ejemplo de la descomposición de una señal en una suma de señales sinusoidales. Como señal original tendremos $cos(t) + 0.5sin(2\pi 0.1t)$. Es decir, hemos tomado un coseno con $\omega = 1$ y $A = 1$, y un seno con $A = 0.5$ y $\nu = 0.1$. La podemos ver en la figura B.1. Según la serie de Fourier, esta señal se puede descomponer en otras señales sinusoidales, que para este caso serán el coseno por un lado y el seno por el otro. Estas dos señales las podemos ver en las figuras B.2 y B.3 respectivamente.

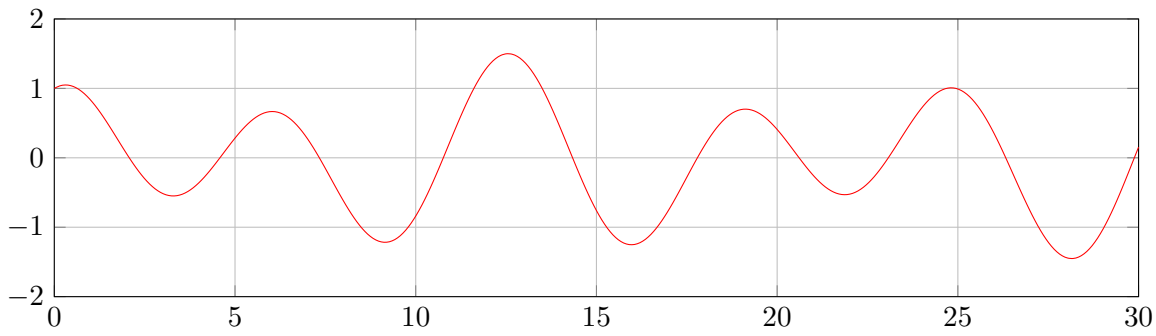


Figura B.1: Función original. $cos(t) + 0.5sin(2\pi 0.1t)$

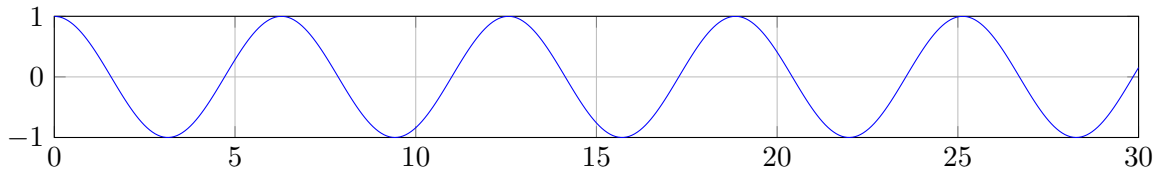


Figura B.2: $cos(t)$

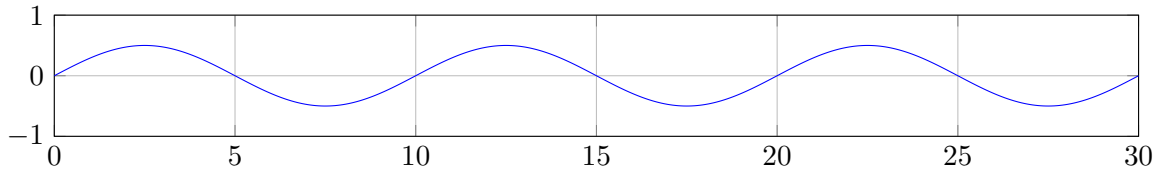


Figura B.3: $0.5sin(2\pi 0.1t)$

La transformada es la aplicación de la serie de Fourier para conseguir una salida continua, en vez de una discreta. Es por ello que el resultado de calcular la transformada de Fourier de una señal es una función, en vez de los coeficientes de la serie. La ecuación de la transformada de Fourier de una función $f(x)$ se puede escribir como

$$F(\xi)\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{+\infty} f(x)e^{-i\xi x}dt \quad . \tag{B.2}$$

La transformada inversa es

$$F(\xi)\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{+\infty} f(x)e^{i\xi x}dt \quad . \tag{B.3}$$

Para el caso que nos interesa, transformar una señal del dominio del tiempo al dominio de la frecuencia, cambiamos ξ por $\omega = 2\pi\nu$ y x por t

$$F(\omega) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{+\infty} f(t)e^{-i\omega t}dt \quad . \tag{B.4}$$

La variable compleja por la que se multipla $f(t)$ se obtiene haciendo uso de la fórmula de

Euler, que relaciona el número e y la variable compleja i con el seno y el coseno. La fórmula se puede escribir como

$$e^{i\omega t} = cos(\omega t) + isen(\omega t) \tag{B.5}$$

y

$$e^{-i\omega t} = cos(\omega t) - isen(\omega t) \quad . \tag{B.6}$$

La transformada de Fourier permite obtener una función que describe todas las funciones sinusoidales que componen una señal periódica (frecuencia, amplitud y fase) y que puede ser dibujada en una gráfica. En la figura B.4 tenemos una señal sintética y su correspondiente transformada de Fourier, donde se pueden ver cuatro picos que corresponden con las frecuencias de las señales que componen la señal original.

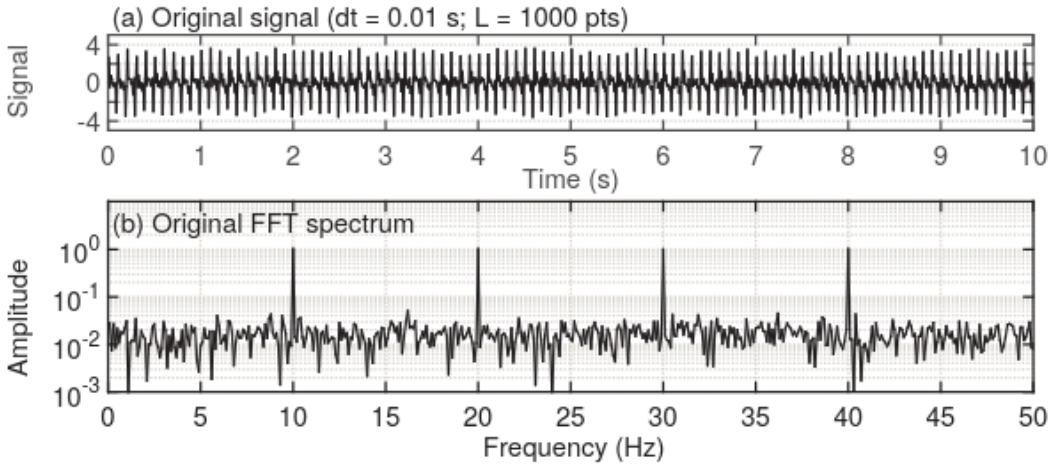


Figura B.4: (a)Señal sintética y (b) el resultado de su FFT. La señal consiste en cuatro señales sinusoidales, con amplitud 1 y frecuencias $f_1 = 10Hz$, $f_2 = 20Hz$, $f_3 = 30Hz$ y $f_4 = 40Hz$

B.2. Transformada Rápida de Fourier

La Transformada rápida de Fourier (*Fast Fourier Transform*, FFT) es un algoritmo que permite calcular la transformada de Fourier discreta (*Discrete Fourier Transform*, DFT) y su inversa. La DFT es la aplicación de la transformada de Fourier a una señal discreta y es el algoritmo principal que se utiliza en el tratamiento de señales en radioastronomía, donde las antenas captan las señales del cielo de forma discreta, tomando una muestra cada cierto tiempo. En realidad existen dos algoritmos dependiendo de la cantidad de muestras que tengamos. Por un lado está la FFT, que sólo puede ser utilizada cuando tenemos una cantidad de muestras que sea potencia de dos. Y por otro lado está la DFT, que se puede aplicar a cualquier tamaño de datos.

Sea x_n una señal periódica discreta en el tiempo. La DFT de esta señal se define como

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}, \qquad k = 0, ..., N-1 \tag{B.7}$$

en la cual X_k es un conjunto de números complejos. La complejidad de esta fórmula es de $O(N^2)$, pero con un algoritmo FFT se puede obtener el mismo resultado con una complejidad de $O(NlogN)$.

La idea principal que siguen todos los algoritmos es la división recurrente de las muestras iniciales hasta conseguir conjuntos de dos muestras. La transformada se realiza sobre estos pares de muestras y se van agrupando en otras de nivel superior que deben resolverse de nuevo hasta llegar al nivel más alto. De forma semejante a lo que ocurre con la transformada de Fourier, la transformada inversa discreta de Fourier se consigue cambiando el signo de la exponencial compleja, con el agregado de un factor $1/N$:

$$X_k = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{\frac{2\pi i}{N} kn}, \qquad k = 0, ..., N-1 \tag{B.8}$$

Por lo general, tenemos que

$$x_n = IDFT\{X_k\} = \frac{1}{N}(DFT\{X_k^*\})^* \tag{B.9}$$

donde el símbolo de asterisco denota la conjugada completa.

El algoritmo de la FFT fue popularizado por los matemáticos estadounidenses James William Cooley y John Wilder Tukey en 1968 siendo el más utilizado en la actualidad [10].

Bibliografía

[1] A. Whitney, R. Cappallo, W. Aldrich, B. Anderson, A. Bos, J. Casse, J. Goodman, S. Parsley, S. Pogrebenko, R. Schilizzi *et al.*, "Mark 4 VLBI correlator: Architecture and algorithms," *Radio Science*, vol. 39, no. 1, pp. 1–24, 2004.

[2] R. A. Primiani, K. H. Young, A. Young, N. Patel, R. W. Wilson, L. Vertatschitsch, B. B. Chitwood, R. Srinivasan, D. MacMahon, and J. Weintroub, "SWARM: a 32 ghz correlator and VLBI beamformer for the submillimeter array," *Journal of Astronomical Instrumentation*, vol. 5, no. 04, p. 1641006, 2016.

[3] J. Hargreaves, "UniBoard: generic hardware for radio astronomy signal processing," in *Millimeter, Submillimeter, and Far-Infrared Detectors and Instrumentation for Astronomy VI*, vol. 8452. International Society for Optics and Photonics, 2012, p. 84522M.

[4] K. Lambeck, "Methods and geophysical applications of satellite geodesy," *Reports on Progress in Physics*, vol. 42, no. 4, p. 547, 1979.

[5] W.-M. Zheng, X.-Z. Zhang, and F.-C. Shu, "CVN harddisk system and software correlator in e-vlbi experiments," *Progress in Astronomy*, vol. 23, no. 3, pp. 272–286, 2005.

[6] A. T. Deller, S. Tingay, M. Bailes, and C. West, "Difx: a software correlator for very long baseline interferometry using multiprocessor computing environments," *Publications of the Astronomical Society of the Pacific*, vol. 119, no. 853, p. 318, 2007.

[7] T. Hobiger, M. Kimura, K. Takefuji, T. Oyama, Y. Koyama, T. Kondo, T. Gotoh, and J. Amagai, "GPU based software correlators-perspectives for VLBI2010," 2010.

[8] F. Zhang, C. Zhao, S. Han, F. Ma, and D. Xiang, "GPU-Based Parallel Implementation of VLBI Correlator for Deep Space Exploration System," *Remote Sensing*, vol. 13, no. 6, p. 1226, 2021.

[9] D. C. Champeney, *Fourier Transforms and Their Physical Applications.* London: Academic Press, 1973.

[10] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[11] J. W. Brown and R. V. Churchill, *Fourier Series and Boundary Value Problems, 5th ed.* New York: McGraw-Hill, 1993.

[12] L. Cohen, "The generalization of the wiener-khinchin theorem," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, vol. 3. IEEE, 1998, pp. 1577–1580.

[13] A. R. Thompson, J. M. Moran, and G. W. Swenson, *Interferometry and synthesis in radio astronomy*. Springer Nature, 2017.

[14] J. G. García, "A DiFX Correlator and advanced polarimetry algorithms for VLBI as Spanish contributions to the EUVGOS project," Master's thesis, Universidad de Alcalá, 2020.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá